Nanodegree Data Scientist

Aprendizado Supervisionado

Projeto: Encontrando doadores para a *CharityML*

Seja bem-vindo ao segundo projeto do Nanodegree Engenheiro de Machine Learning! Neste notebook, você receberá alguns códigos de exemplo e será seu trabalho implementar as funcionalidades adicionais necessárias para a conclusão do projeto. As seções cujo cabeçalho começa com **'Implementação'** indicam que o bloco de código posterior requer funcionalidades adicionais que você deve desenvolver. Para cada parte do projeto serão fornecidas instruções e as diretrizes da implementação estarão marcadas no bloco de código com uma expressão `'TODO'`. Por favor, leia cuidadosamente as instruções!

Além de implementações de código, você terá de responder questões relacionadas ao projeto e à sua implementação. Cada seção onde você responderá uma questão terá um cabeçalho com o termo **'Questão X'**. Leia com atenção as questões e forneça respostas completas nas caixas de texto que começam com o termo **'Resposta:'**. A submissão do seu projeto será avaliada baseada nas suas respostas para cada uma das questões além das implementações que você disponibilizar.

> **Nota:** Por favor, especifique QUAL A VERSÃO DO PYTHON utilizada por você para a submissão deste notebook. As células "Code" e "Markdown" podem ser executadas utilizando o atalho do teclado **Shift + Enter**. Além disso, as células "Markdown" podem ser editadas clicando-se duas vezes na célula.
>
> **Toda implementação foi realizada utilizando-se a linguagem `'Python 3'` e `scikit-learn v.0.20.0`**

Iniciando

Neste projeto, você utilizará diversos algoritmos de aprendizado supervisionado para modelar com precisão a remuneração de indivíduos utilizando dados coletados no censo americano de 1994. Você escolherá o algoritmo mais adequado através dos resultados preliminares e irá otimizá-lo para modelagem dos dados. O seu objetivo com esta implementação é construir um modelo que pode predizer com precisão se um indivíduo possui uma remuneração superior a $50,000. Este tipo de tarefa pode surgir em organizações sem fins lucrativos que sobrevivem de doações. Entender a remuneração de um indivíduo pode ajudar a organização o montante mais adequado para uma solicitação de doação, ou ainda se eles realmente deveriam entrar em contato com a pessoa. Enquanto pode ser uma tarefa difícil determinar a faixa de renda de uma pesssoa de maneira direta, nós podemos inferir estes valores através de outros recursos disponíveis publicamente.

O conjunto de dados para este projeto se origina do Repositório de Machine Learning UCI (https://archive.ics.uci.edu/ml/datasets/Census+Income) e foi cedido por Ron Kohavi e Barry Becker, após a sua publicação no artigo *"Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid"*. Você pode encontrar o artigo de Ron Kohavi online (https://www.aaai.org/Papers/KDD/1996/KDD96-033.pdf). Os dados que investigaremos aqui possuem algumas pequenas modificações se comparados com os dados originais, como por exemplo a remoção da funcionalidade `'fnlwgt'` e a remoção de registros inconsistentes.

Explorando os dados

Execute a célula de código abaixo para carregas as bibliotecas Python necessárias e carregas os dados do censo. Perceba que a última coluna deste cojunto de dados, `'income'`, será o rótulo do nosso alvo (se um indivíduo possui remuneração igual ou maior do que $50,000 anualmente). Todas as outras colunas são dados de cada indívduo na base de dados do censo.

In [1]:

```
# Importe as bibliotecas necessárias para o projeto.
import numpy as np
import pandas as pd
from time import time
from IPython.display import display # Permite a utilização da função display() para DataFrames.

# Importação da biblioteca de visualização visuals.py
import visuals as vs

# Exibição amigável para notebooks
%matplotlib inline

# Carregando os dados do Censo
data = pd.read_csv("census.csv")

# Sucesso - Exibindo o primeiro registro
display(data.head(n=1))
```

	age	workclass	education_level	education-num	marital-status	occupation	relationship	
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	W

Implementação: Explorando os Dados

Uma investigação superficial da massa de dados determinará quantos indivíduos se enquadram em cada grupo e nos dirá sobre o percentual destes indivúdos com remuneração anual superior a $50,000. No código abaixo, você precisará calcular o seguinte:

- O número total de registros, `'n_records'`
- O número de indivíduos com remuneração anual superior a $50,000, `'n_greater_50k'`.
- O número de indivíduos com remuneração anual até $50,000, `'n_at_most_50k'`.
- O percentual de indivíduos com remuneração anual superior a $50,000, `'greater_percent'`.

DICA: Você pode precisar olhar a tabela acima para entender como os registros da coluna `'income'` estão formatados.

In [2]:

```
# TODO: Número total de registros.
n_records = len(data)

# TODO: Número de registros com remuneração anual superior à $50,000
n_greater_50k = len(data[data['income']=='>50K'])


# TODO: O número de registros com remuneração anual até $50,000
n_at_most_50k = len(data[data['income']=='<=50K'])

# data[data['income']=='<=50K'].income.count()
# também traz o mesmo resultado

# Verificação:
#data[data['income']=='>50K'].income.count() + data[data['income']=='<=50K'].income.cou
nt() == data['income'].count()

# TODO: O percentual de indivíduos com remuneração anual superior à $50,000
greater_percent = n_greater_50k/n_records

# Exibindo os resultados
# Inclui parêntesis por questões de sintaxe e multipliquei greater_percent por 100
# devido ao formato percentual
print("Total number of records: {}".format(n_records))
print("Individuals making more than $50,000: {}".format(n_greater_50k))
print("Individuals making at most $50,000: {}".format(n_at_most_50k))
print("Percentage of individuals making more than $50,000: {:.2f}%".format(100*greater_
percent))
```

```
Total number of records: 45222
Individuals making more than $50,000: 11208
Individuals making at most $50,000: 34014
Percentage of individuals making more than $50,000: 24.78%
```

Explorando as colunas

- **age**: contínuo.
- **workclass**: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
- **education**: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
- **education-num**: contínuo.
- **marital-status**: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
- **occupation**: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
- **relationship**: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
- **race**: Black, White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other.
- **sex**: Female, Male.
- **capital-gain**: contínuo.
- **capital-loss**: contínuo.
- **hours-per-week**: contínuo.
- **native-country**: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

Preparando os dados

Antes de que os dados possam ser utilizados como input para algoritmos de machine learning, muitas vezes eles precisam ser tratados, formatados e reestruturados — este processo é conhecido como **pré-processamento**. Felizmente neste conjunto de dados não existem registros inconsistentes para tratamento, porém algumas colunas precisam ser ajustadas. Este pré-processamento pode ajudar muito com o resultado e poder de predição de quase todos os algoritmos de aprendizado.

Transformando os principais desvios das colunas contínuas

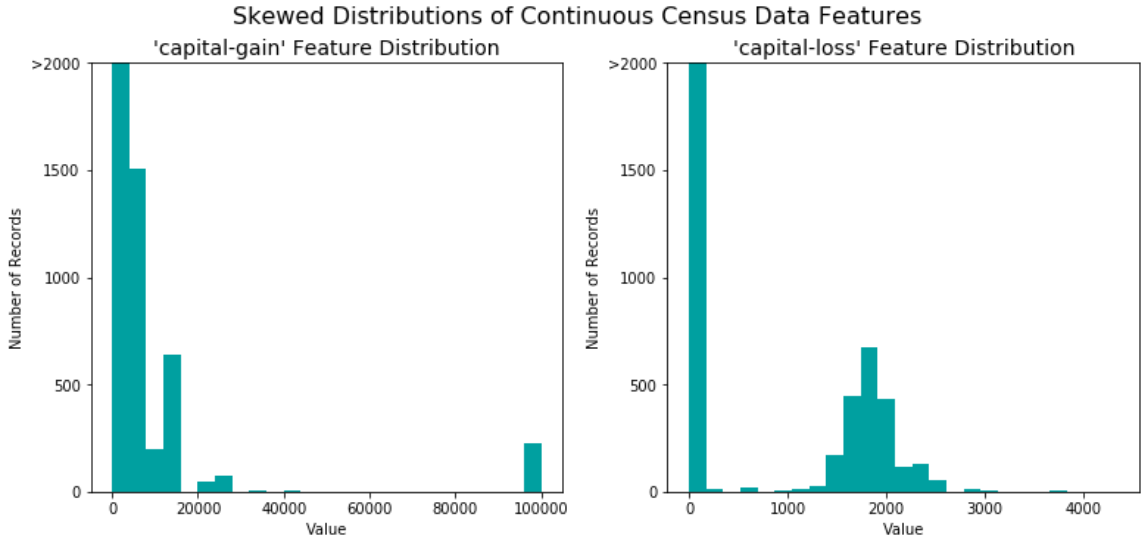
Um conjunto de dados pode conter ao menos uma coluna onde os valores tendem a se próximar para um único número, mas também podem conter registros com o mesmo atributo contendo um valor muito maior ou muito menor do que esta tendência. Algoritmos podem ser sensíveis para estes casos de distribuição de valores e este fator pode prejudicar sua performance se a distribuição não estiver normalizada de maneira adequada. Com o conjunto de dados do censo, dois atributos se encaixam nesta descrição: '`capital-gain`' e '`capital-loss`'.

Execute o código da célula abaixo para plotar um histograma destes dois atributos. Repare na distribuição destes valores.

In [3]:

```
# Dividindo os dados entre features e coluna alvo
income_raw = data['income']
features_raw = data.drop('income', axis = 1)

# Visualizando os principais desvios das colunas contínuas entre os dados
vs.distribution(data)
```



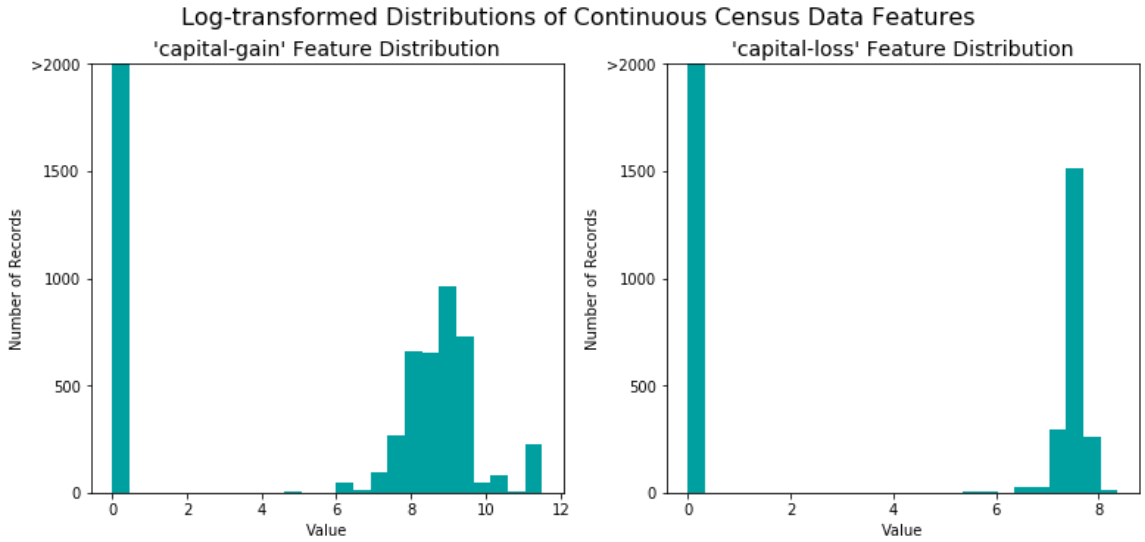
Para atributos com distribuição muito distorcida, tais como `'capital-gain'` e `'capital-loss'`, é uma prática comum aplicar uma transformação logarítmica (https://en.wikipedia.org/wiki/Data_transformation_(statistics)) nos dados para que os valores muito grandes e muito pequenos não afetem a performance do algoritmo de aprendizado. Usar a transformação logarítmica reduz significativamente os limites dos valores afetados pelos outliers (valores muito grandes ou muito pequenos). Deve-se tomar cuidado ao aplicar esta transformação, pois o logaritmo de `0` é indefinido, portanto temos que incrementar os valores em uma pequena quantia acima de `0` para aplicar o logaritmo adequadamente.

Execute o código da célula abaixo para realizar a transformação nos dados e visualizar os resultados. De novo, note os valores limite e como os valores estão distribuídos.

In [4]:

```
# Aplicando a transformação de log nos registros distorcidos.
skewed = ['capital-gain', 'capital-loss']
features_log_transformed = pd.DataFrame(data = features_raw)
features_log_transformed[skewed] = features_raw[skewed].apply(lambda x: np.log(x + 1))

# Visualizando as novas distribuições após a transformação.
vs.distribution(features_log_transformed, transformed = True)
```



Normalizando atributos numéricos

Além das transformações em atributos distorcidos, é uma boa prática comum realizar algum tipo de adaptação de escala nos atributos numéricos. Ajustar a escala nos dados não modifica o formato da distribuição de cada coluna (tais como `'capital-gain'` ou `'capital-loss'` acima); no entanto, a normalização garante que cada atributo será tratado com o mesmo peso durante a aplicação de aprendizado supervisionado. Note que uma vez aplicada a escala, a observação dos dados não terá o significado original, como exemplificado abaixo.

Execute o código da célula abaixo para normalizar cada atributo numérico, nós usaremos para isso a [`sklearn.preprocessing.MinMaxScaler`](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html).

In [5]:

```
# Importando sklearn.preprocessing.StandardScaler
from sklearn.preprocessing import MinMaxScaler

# Inicializando um aplicador de escala e aplicando em seguida aos atributos
scaler = MinMaxScaler() # default=(0, 1)
numerical = ['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week']

features_log_minmax_transform = pd.DataFrame(data= features_log_transformed)
features_log_minmax_transform[numerical] = scaler.fit_transform(features_log_transforme
d[numerical])

# Exibindo um exemplo de registro com a escala aplicada
display(features_log_minmax_transform.head(n=5))
```

	age	workclass	education_level	education-num	marital-status	occupation	relationshi
0	0.301370	State-gov	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-famil
1	0.452055	Self-emp-not-inc	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband
2	0.287671	Private	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-famil
3	0.493151	Private	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband
4	0.150685	Private	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wife

Implementação: Pré-processamento dos dados

A partir da tabela em **Explorando os dados** acima, nós podemos observar que existem diversos atributos não-numéricos para cada registro. Usualmente, algoritmos de aprendizado esperam que os inputs sejam numéricos, o que requer que os atributos não numéricos (chamados de *variáveis de categoria*) sejam convertidos. Uma maneira popular de converter as variáveis de categoria é utilizar a estratégia **one-hot encoding**. Esta estratégia cria uma variável para cada categoria possível de cada atributo não numérico. Por exemplo, assuma que `algumAtributo` possuí três valores possíveis: `A`, `B`, ou `C`. Nós então transformamos este atributo em três novos atributos: `algumAtributo_A`, `algumAtributo_B` e `algumAtributo_C`.

	algumAtributo		**algumAtributo_A**	**algumAtributo_B**	**algumAtributo_C**
0	B		0	1	0
1	C	----> one-hot encode ---->	0	0	1
2	A		1	0	0

Além disso, assim como os atributos não-numéricos, precisaremos converter a coluna alvo não-numérica, `'income'`, para valores numéricos para que o algoritmo de aprendizado funcione. Uma vez que só existem duas categorias possíveis para esta coluna ("<=50K" e ">50K"), nós podemos evitar a utilização do one-hot encoding e simplesmente transformar estas duas categorias para `0` e `1`, respectivamente. No trecho de código abaixo, você precisará implementar o seguinte:

- Utilizar `pandas.get_dummies()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html?highlight=get_dummies#pandas.get_dummies) para realizar o one-hot encoding nos dados da `'features_log_minmax_transform'`.
- Converter a coluna alvo `'income_raw'`
  - Transforme os registros com "<=50K" para `0` e os registros com ">50K" para `1`.

In [6]:

```
# TODO: Utilize o one-hot encoding nos dados em 'features_log_minmax_transform' utiliza
ndo pandas.get_dummies()
features_final = pd.get_dummies(features_log_minmax_transform)

# TODO: Faça o encode da coluna 'income_raw' para valores numéricos

# Cria um array de zeros com o mesmo tamanho de 'income_raw' e preenche-o com 1
# nas posições em que 'income' é '>50K'
income = np.zeros(shape= (len(income_raw)))
for i in range(len(income_raw)):
    if (income_raw[i] == '>50K'):
        income[i] = 1

# Converte o tipo de objeto de 'array' para 'Serie'.
# Não é estritamente necessário, mas conserva o tipo de objeto previamente utilizado
income = pd.Series(income)

# income = income_raw.apply(lambda x: 1 if x == '>50K' else 0)
# também traz o mesmo resultado, de modo (bem) mais simplificado

# Exiba o número de colunas depois do one-hot encoding
encoded = list(features_final.columns)
print("{} total features after one-hot encoding.".format(len(encoded)))

# Descomente a linha abaixo para ver as colunas após o encode
#print(encoded)
```

```
103 total features after one-hot encoding.
```

Embaralhar e dividir os dados

Agora todas as *variáveis de categoria* foram convertidas em atributos numéricos e todos os atributos numéricos foram normalizados. Como sempre, nós agora dividiremos os dados entre conjuntos de treinamento e de teste. 80% dos dados serão utilizados para treinamento e 20% para teste.

Execute o código da célula abaixo para realizar divisão.

In [7]:

```
# Importar train_test_split
# from sklearn.cross_validation import train_test_split

# Alterei o módulo "cross_validation" por "model_selection" por questões de obsolescênc
ia
from sklearn.model_selection import train_test_split

# Dividir os 'atributos' e 'income' entre conjuntos de treinamento e de testes.
X_train, X_test, y_train, y_test = train_test_split(features_final,
                                                    income,
                                                    test_size= 0.2,
                                                    random_state= 0)

# Show the results of the split
print("Training set has {} samples.".format(X_train.shape[0]))
print("Testing set has {} samples.".format(X_test.shape[0]))
```

```
Training set has 36177 samples.
Testing set has 9045 samples.
```

Avaliando a performance do modelo

Nesta seção, nós investigaremos quatro algoritmos diferentes e determinaremos qual deles é melhor para a modelagem dos dados. Três destes algoritmos serão algoritmos de aprendizado supervisionado de sua escolha e o quarto algoritmo é conhecido como *naive predictor*.

Métricas e o Naive predictor

CharityML, equipada com sua pesquisa, sabe que os indivíduos que tem mais do que $50,000 possuem maior probabilidade de doar para a sua campanha de caridade. Por conta disto, a *CharityML* está particularmente interessada em predizer com acurácia quais indivíduos possuem remuneração acima de $50,000. Parece que utilizar **acurácia (accuracy)** como uma métrica para avaliar a performance de um modelo é um parâmetro adequado. Além disso, identificar alguém que *não possui* remuneração acima de $50,000 como alguém que recebe acima deste valor seria ruim para a *CharityML*, uma vez que eles estão procurando por indivíduos que desejam doar. Com isso, a habilidade do modelo em predizer com precisão aqueles que possuem a remuneração acima dos $50,000 é *mais importante* do que a habilidade de realizar o **recall** destes indivíduos. Nós podemos utilizar a fórmula **F-beta score** como uma métrica que considera ambos: precision e recall.

$$F_{\beta} = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

Em particular, quando $\beta = 0.5$, maior ênfase é atribuída para a variável precision. Isso é chamado de **$F_{0.5}$ score** (ou F-score, simplificando).

Analisando a distribuição de classes (aqueles que possuem remuneração até $50,000 e aqueles que possuem remuneração superior), fica claro que a maioria dos indivíduos não possui remuneração acima de $50,000. Isto pode ter grande impacto na **acurácia (accuracy)**, uma vez que nós poderíamos simplesmente dizer *"Esta pessoa não possui remuneração acima de $50,000"* e estar certos em boa parte das vezes, sem ao menos olhar os dados! Fazer este tipo de afirmação seria chamado de **naive**, uma vez que não consideramos nenhuma informação para balizar este argumento. É sempre importante considerar a *naive prediction* para seu conjunto de dados, para ajudar a estabelecer um *benchmark* para análise da performance dos modelos. Com isso, sabemos que utilizar a *naive prediction* não traria resultado algum: Se a predição apontasse que todas as pessoas possuem remuneração inferior a $50,000, a *CharityML* não identificaria ninguém como potencial doador.

Nota: Revisando: accuracy, precision e recall

Accuracy mede com que frequência o classificador faz a predição correta. É a proporção entre o número de predições corretas e o número total de predições (o número de registros testados).

Precision informa qual a proporção de mensagens classificamos como spam eram realmente spam. Ou seja, é a proporção de verdadeiros positivos (mensagens classificadas como spam que eram realmente spam) sobre todos os positivos (todas as palavras classificadas como spam, independente se a classificação estava correta), em outras palavras, é a proporção

`[Verdadeiros positivos/(Verdadeiros positivos + Falso positivos)]`

Recall (sensibilidade) nos informa qual a proporção das mensagens que eram spam que foram corretamente classificadas como spam. É a proporção entre os verdadeiros positivos (classificados como spam, que realmente eram spam) sobre todas as palavras que realmente eram spam. Em outras palavras, é a proporção entre

`[Verdadeiros positivos/(Verdadeiros positivos + Falso negativos)]`

Para problemas de classificação distorcidos em suas distribuições, como no nosso caso, por exemplo, se tivéssemos 100 mensagens de texto, apenas 2 fossem spam e todas as outras não fossem, a *accuracy* por si só não seria uma métrica tão boa. Nós poderiamos classificar 90 mensagems como "não-spam" (incluindo as 2 que eram spam mas que teriam sido classificadas como não-spam e, por tanto, seriam falso negativas)

e 10 mensagems como spam (todas as 10 falso positivas) e ainda assim teriamos uma boa pontuação de accuracy. Para estes casos, precision e recall são muito úteis. Estas duas métricas podem ser combinadas para resgatar o F1 score, que é calculado através da média (harmônica) dos valores de precision e de recall. Este score pode variar entre 0 e 1, sendo 1 o melhor resultado possível para o F1 score.(Consideramos a média harmônica pois estamos lidando com proporções).

Questão 1 - Performance do Naive Predictor

- Se escolhessemos um modelo que sempre prediz que um indivíduo possui remuneração acima de $50,000, qual seria a accuracy e o F-score considerando este conjunto de dados? Você deverá utilizar o código da célula abaixo e atribuir os seus resultados para as variáveis `'accuracy'` e `'fscore'` que serão usadas posteriormente.

Por favor, note que o propósito ao gerar um naive predictor é simplesmente exibir como um modelo sem nenhuma inteligência se comportaria. No mundo real, idealmente o seu modelo de base será o resultado de um modelo anterior ou poderia ser baseado em um *paper* no qual você se basearia para melhorar. Quando não houver qualquer *benchmark* de modelo, utilizar um *naive predictor* será melhor do que uma escolha aleatória.

DICA:

- Quando temos um modelo que sempre prediz '1' (ex. o indivíduo possui remuneração superior à 50k) então nosso modelo não terá Verdadeiros Negativos ou Falso Negativos, pois nós não estaremos afirmando que qualquer dos valores é negativo (ou '0') durante a predição. Com isso, nossa *accuracy* neste caso se torna o mesmo valor da *precision* (Verdadeiros positivos/ (Verdadeiros positivos + Falso positivos)) pois cada predição que fizemos com o valor '1' que deveria ter o valor '0' se torna um falso positivo; nosso denominador neste caso é o número total de registros.
- Nossa pontuação de *Recall* (Verdadeiros positivos/(Verdadeiros Positivos + Falsos negativos)) será 1 pois não teremos Falsos negativos.

In [8]:

```
'''
TP = np.sum(income) # Contando pois este é o caso "naive". Note que 'income' são os dad
os 'income_raw' convertidos
para valores numéricos durante o passo de pré-processamento de dados.
FP = income.count() - TP # Específico para o caso naive

TN = 0 # Sem predições negativas para o caso naive
FN = 0 # Sem predições negativas para o caso naive
'''
# TODO: Calcular accuracy, precision e recall
# Para o naive predictor que sempre classifica como 1, temos:
TP = np.sum(income)
FP = len(income) - TP
TN = 0
FN = 0
T = TP + TN + FP + FN

accuracy = (TP + TN)/T
recall = TP/(TP + FN)
precision = TP/(TP + FP)

# TODO: Calcular o F-score utilizando a fórmula acima para o beta = 0.5 e os valores co
rretos de precision e recall.
beta = .5
fscore = (1 + beta**2)*(precision * recall)/((beta**2*precision) + recall)

# Exibir os resultados
print("Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".format(accuracy, fsc
ore))
```

```
Naive Predictor: [Accuracy score: 0.2478, F-score: 0.2917]
```

Modelos de Aprendizado Supervisionado

Estes são alguns dos modelos de aprendizado supervisionado disponíveis em scikit-learn (http://scikit-learn.org/stable/supervised_learning.html)

- Gaussian Naive Bayes (GaussianNB)
- Decision Trees (Árvores de decisão)
- Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting)
- K-Nearest Neighbors (KNeighbors)
- Stochastic Gradient Descent Classifier (SGDC)
- Support Vector Machines (SVM)
- Logistic Regression

Questão 2 - Aplicação do Modelo

Liste três dos modelos de aprendizado supervisionado acima que são apropriados para este problema que você irá testar nos dados do censo. Para cada modelo escolhido

- Descreva uma situação do mundo real onde este modelo pode ser utilizado.
- Quais são as vantagens da utilização deste modelo; quando ele performa bem?
- Quais são as fraquezas do modelo; quando ele performa mal?
- O que torna este modelo um bom candidato para o problema, considerando o que você sabe sobre o conjunto de dados?

DICA:

Estruture sua resposta no mesmo formato acima, com 4 partes para cada um dos modelos que você escolher. Por favor, inclua referências em cada uma das respostas.

Resposta:

Guiando-se pelo excelente fluxograma fornecido pela equipe do `scikit-learn` [1] (https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html), para a escolha de bons modelos supervisionados de classificação binária, estando disponíveis menos de 100 mil amostras para treino e teste, somos direcionados a *Support Vector Machines* (Máquinas de Vetores-Suporte), *Nearest Neighbors* (Vizinhos Mais Próximos), *Ensemble Methods* (Métodos de Conjuntos) e *Generalized Linear Models* (Modelos Lineares Generalizados).

Dada a variedade de opções, foi feita uma pré-seleção testando-se vários modelos disponíveis no repositório do `sklearn` e observando-se atributos como o tempo de treinamento e a função média harmônica ponderada entre precisão e *recall* dos candidatos (*f-beta score*). *[Vide Seção "Testes Extras" no final]*

Os algoritmos mais promissores foram:

- *Gradient Boosting Classifier*, da classe *Ensemble Methods*
- *AdaBoost Classifier*, também da classe *Ensemble Methods*
- *Logistic Regression*, da classe *Generalized Linear Models*

Gradient Boosting Classifier, por se tratar de um *Boosting Ensemble Method*, tem por objetivo combinar vários classificadores "fracos" para produzir um conjunto classificador "forte". Construído com base em um algoritmo de aprendizado para melhorar a generalidade e robustez se comparado a um classificador único, seus estimadores-base são montados sequencialmente para evitar que sua combinação induza enviesamentos. Trata-se de uma generalização dos métodos impulsionados para funções diferenciáveis de perdas (e.g. *log-likelihood* binomial, perda exponencial etc.)

- *Aplicação prática*: Modelos *Gradient Boosting* são utilizados em várias áreas, incluído ranqueamento de buscas na internet e ecologia [2] (https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting).
- *Vantagens*: É um algoritmo que lida naturalmente com dados de características heterogêneas, tem alto poder preditivo e é robusto a *outliers* no espaço de saída, graças à sua adequação a funções robustas de perdas.
- *Desvantagens*: Escalabilidade pode ser um problema devido a sua natureza sequencial de impulsionamento, dificultando o processamento em paralelo. Também leva mais tempo durante a etapa de treino.
- *O que torna este modelo um bom candidato ao problema:* Pela sua reconhecida boa performance e pela quantidade de dados suficiente e limpa - se lida bem com dados "heterogêneos", não terá dificuldade com dados "homogêneos" - , este é um modelo que deve ser experimentado.

AdaBoost, apresentando em 1995 por Freund e Schapire [3], seu princípio também é adequar uma sequência de classificadores fracos em repetidas versões modificadas dos dados, combinar esta sequência em uma soma (ou "votação") ponderada e produzir a predição final. A cada iteração, mais peso é dado para as saídas erroneamente classificadas, forçando o algoritmo a focar nas características mais difíceis de se acertar para incrementar a performance global do modelo

- *Aplicação prática*: Na indústria, algoritmos *boosting* tem sido utilizados no problema de classificação binária para detecção de faces, tendo o algoritmo que identificar se uma porção da imagem é um rosto ou parte do plano de fundo [4] (https://en.wikipedia.org/w/index.php?title=Viola%E2%80%93Jones_object_detection_framework&oldid=863133691)
- *Vantagens*: É um algoritmo que costuma ser muito adaptativo, capturar fronteiras complexas de decisão e não requerer muitos ajustes de parâmetros.

- *Desvantagens*: Por outro lado, na ausência de pré-processamento dos dados, ruídos e *outliers* podem impactar negativamente sua performance. Além disso, se um modelo complexo for utilizado como classificador-base, pode haver *overfitting* (sobretreinamento) nos dados de treino e demora tanto nesta etapa quanto na de predição.
- *O que torna este modelo um bom candidato ao problema:* Nosso conjunto de dados é amplo e limpo, de forma a possibilitar que iterações múltiplas sejam rapidamente realizadas para maximizar a acurácia nos dados inéditos de teste. O modelo pode demorar para ser treinado, porém seu treinamento seria periódico e não *on-line*, o que não faz disto um problema.

Regressão Logística, a despeito do nome, é um modelo linear utilizado para classificação ao invés de regressão, sendo também encontrado na literatura como *"Regressão Logit"* e *"Classificador de Máxima Entropia"*. Neste modelo, as probabilidades descrevendo as saídas possíveis (0 ou 1) são modeladas de acordo com a função logística $p(x) = \frac{1}{1+e^{-b^T \cdot \vec{x}}}$. Convenientemente, pelas propriedades de derivação de $p(x)$, podemos obter as *odds* (chance de ocorrência dividida pela chance de não ocorrência) calculando-se $\frac{p(x)}{1-p(x)}$ [5] (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)

- *Aplicação prática*: É amplamente utilizada em problemas de classificação binária, comumente predizendo se um cliente comprará ou não determinado produto ou se determinado grupo oferece ou não risco à concessão de créditos [6] (https://pt.wikipedia.org/wiki/Regressão_logística).
- *Vantagens*: É rápido nas etapas de treino e predição, dando bons resultados mesmo com poucas características.
- *Desvantagens*: Por ser da classe de modelos lineares generalizados, assume fronteiras lineares para classificação, as quais são insuficientes para uma boa segregação de características com disposições complexas no espaço destes vetores.
- *O que torna este modelo um bom candidato ao problema:* O problema da *CharityML* consiste em uma classificação binária com dados limpos, duas condições altamente favoráveis para se empregar regressões logísticas.

Referências

[1] CHOOSING THE RIGHT ESTIMATOR. In: Scikit-learn: Machine Learning in {P}ython. Disponível em: https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html (https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html). Acesso em: 25 nov. 2018

[2] GRADIENT BOOSTING. In: Scikit-learn: Machine Learning in {P}ython. Disponível em: https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting (https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting). Acesso em: 25 nov. 2018

[3] Y. Freund, and R. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”, 1997

[4] Wikipedia contributors. (2018, October 8). Viola–Jones object detection framework. In Wikipedia, The Free Encyclopedia. Retrieved 16:49, November 25, 2018, from https://en.wikipedia.org/w/index.php?title=Viola%E2%80%93Jones_object_detection_framework&oldid=863133691 (https://en.wikipedia.org/w/index.php?title=Viola%E2%80%93Jones_object_detection_framework&oldid=863133691)

[5] LOGISTIC REGRESSION. In: Scikit-learn: Machine Learning in {P}ython. Disponível em: https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression). Acesso em: 25 nov. 2018

[6] REGRESSÃO LOGÍSTICA. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2018. Disponível em: https://pt.wikipedia.org/w/index.php?title=Regress%C3%A3o_log%C3%ADstica&oldid=53027525 (https://pt.wikipedia.org/w/index.php?title=Regress%C3%A3o_log%C3%ADstica&oldid=53027525). Acesso em: 25 nov. 2018.

Implementação - Criando um Pipeline de Treinamento e Predição

Para avaliar adequadamente a performance de cada um dos modelos que você escolheu é importante que você crie um pipeline de treinamento e predição que te permite de maneira rápida e eficiente treinar os modelos utilizando vários tamanhos de conjuntos de dados para treinamento, além de performar predições nos dados de teste. Sua implementação aqui será utilizada na próxima seção. No bloco de código abaixo, você precisará implementar o seguinte:

- Importar `fbeta_score` e `accuracy_score` de `sklearn.metrics` (http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics).
- Adapte o algoritmo para os dados de treinamento e registre o tempo de treinamento.
- Realize predições nos dados de teste `X_test`, e também nos 300 primeiros pontos de treinamento `X_train[:300]`.
  - Registre o tempo total de predição.
- Calcule a acurácia tanto para o conjundo de dados de treino quanto para o conjunto de testes.
- Calcule o F-score para os dois conjuntos de dados: treino e testes.
  - Garanta que você configurou o parâmetro `beta`!

In [9]:

```
# TODO: Import two metrics from sklearn - fbeta_score and accuracy_score
from sklearn.metrics import fbeta_score, accuracy_score

def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
    '''
    inputs:
       - learner: the learning algorithm to be trained and predicted on
       - sample_size: the size of samples (number) to be drawn from training set
       - X_train: features training set
       - y_train: income training set
       - X_test: features testing set
       - y_test: income testing set
    '''

    results = {}

    # TODO: Fit the learner to the training data using slicing with 'sample_size'
    #       using .fit(training_features[:], training_labels[:])
    start = time() # Get start time
    learner.fit(X_train[:sample_size], y_train[:sample_size])
    end = time() # Get end time


    # TODO: Calculate the training time
    results['train_time'] = end - start

    # TODO: Get the predictions on the test set(X_test),
    #       then get predictions on the first 300 training samples(X_train) using .pred
ict()
    start = time() # Get start time
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train[:300])
    end = time() # Get end time

    # TODO: Calculate the total prediction time
    results['pred_time'] = end - start

    # TODO: Compute accuracy on the first 300 training samples which is y_train[:300]
    results['acc_train'] = accuracy_score(y_train[:300],
                                          predictions_train[:300],
                                          normalize= True,
                                          sample_weight= None)

    # TODO: Compute accuracy on test set using accuracy_score()
    results['acc_test'] = accuracy_score(y_test,
                                         predictions_test,
                                         normalize= True,
                                         sample_weight= None)

    beta = .5
    # TODO: Compute F-score on the the first 300 training samples using fbeta_score()

    results['f_train'] = fbeta_score(y_train[:300],
                                     predictions_train[:300],
                                     beta,
                                     labels= None,
                                     pos_label= 1,
                                     #average= 'binary',
                                     sample_weight= None)
```

```
    # TODO: Compute F-score on the test set which is y_test
    results['f_test'] = fbeta_score(y_test,
                                    predictions_test,
                                    beta,
                                    labels= None,
                                    pos_label= 1,
                                    #average= 'binary',
                                    sample_weight= None)

    # Success
    print("{} trained on {} samples.".format(learner.__class__.__name__, sample_size))

    # Return the results
    return results
```

Implementação: Validação inicial do modelo

No código da célular, você precisará implementar o seguinte:

- Importar os três modelos de aprendizado supervisionado que você escolheu na seção anterior
- Inicializar os três modelos e armazená-los em `'clf_A'`, `'clf_B'`, e `'clf_C'`.
  - Utilize um `'random_state'` para cada modelo que você utilizar, caso seja fornecido.
  - **Nota:** Utilize as configurações padrão para cada modelo - você otimizará um modelo específico em uma seção posterior
- Calcule o número de registros equivalentes à 1%, 10%, e 100% dos dados de treinamento.
  - Armazene estes valores em `'samples_1'`, `'samples_10'`, e `'samples_100'` respectivamente.

Nota: Dependendo do algoritmo de sua escolha, a implementação abaixo pode demorar algum tempo para executar!

In [10]:

```
# TODO: Importe os três modelos de aprendizado supervisionado da sklearn
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression


# TODO: Inicialize os três modelos
clf_A = GradientBoostingClassifier(random_state= 0)
clf_B = AdaBoostClassifier(random_state= 0)
clf_C = LogisticRegression(random_state= 0, solver = 'lbfgs')

# TODO: Calcule o número de amostras para 1%, 10%, e 100% dos dados de treinamento
# HINT: samples_100 é todo o conjunto de treinamento e.x.: len(y_train)
# HINT: samples_10 é 10% de samples_100
# HINT: samples_1 é 1% de samples_100
samples_100 = int(round(    len(y_train),0))
samples_10  = int(round( .1*len(y_train),0))
samples_1   = int(round(.01*len(y_train),0))

# Colete os resultados dos algoritmos de aprendizado
results = {}
for clf in [clf_A, clf_B, clf_C]:
    clf_name = clf.__class__.__name__
    results[clf_name] = {}
    for i, samples in enumerate([samples_1, samples_10, samples_100]):
        results[clf_name][i] = train_predict(clf, samples, X_train, y_train, X_test, y_
test)
```

```
GradientBoostingClassifier trained on 362 samples.
GradientBoostingClassifier trained on 3618 samples.
GradientBoostingClassifier trained on 36177 samples.
AdaBoostClassifier trained on 362 samples.
AdaBoostClassifier trained on 3618 samples.
AdaBoostClassifier trained on 36177 samples.
LogisticRegression trained on 362 samples.
LogisticRegression trained on 3618 samples.
LogisticRegression trained on 36177 samples.
```

In [11]:

```
# Run metrics visualization for the three supervised learning models chosen

# Fiz algumas alterações no script visuals.py para facilitar a visualização dos gráfico
s de benchmarking
# Inativei o atributo pl.tight_layout() e defini pl.subplots(2, 3, figsize = (16,11))
vs.evaluate(results, accuracy, fscore)
```

Melhorando os resultados

Nesta seção final, você irá escolher o melhor entre os três modelos de aprendizado supervisionado para utilizar nos dados estudados. Você irá então realizar uma busca grid para otimização em todo o conjunto de dados de treino (`X_train` e `y_train`) fazendo o tuning de pelo menos um parâmetro para melhorar o F-score anterior do modelo.

Questão 3 - Escolhendo o melhor modelo

- Baseado na validação anterior, em um ou dois parágrafos explique para a *CharityML* qual dos três modelos você acredita ser o mais apropriado para a tarefa de identificar indivíduos com remuneração anual superior à $50,000.

DICA: Analise o gráfico do canto inferior esquerdo da célula acima(a visualização criada através do comando `vs.evaluate(results, accuracy, fscore)`) e verifique o F score para o conjunto de testes quando 100% do conjunto de treino é utilizado. Qual modelo possui o maior score? Sua resposta deve abranger os seguintes pontos:

- métricas - F score no conjunto de testes quando 100% dos dados de treino são utilizados,
- tempo de predição/treinamento
- a adequação do algoritmo para este conjunto de dados.

Resposta:

Dentre os três modelos, o mais adequado para o conjunto de dados da *CharityML* é o **Gradient Boosting Classifier**, já que:

- Apresentou o melhor *f-beta score* dentre os três candidatos, o que significa que entrega bons resultados tanto em precisão quanto em *recall* e teve também, ligeiramente, a melhor acurácia dentre os três modelos nas três amostras de diferentes tamanhos nos dados de teste (*vide tabela abaixo para a amostra de 100% dos dados*);
- *Apesar* de ter apresentado maior tempo de treinamento em todos diferentes tamanhos de amostros e tempo intermediário para predição, como nosso foco não é utilização do modelo como aplicação *online* de treinamento recorrente mas sim sua precisão ao apontar quem tem renda acima de $50,000, devemos valorizar este segundo quesito;
- Adequa-se tanto ao volume quanto ao tipo de dados descritivos e limpos em questão

In [12]:

```
# Dataframe de benchmarking
idx_metrics = ['train_time', 'pred_time', 'acc_train', 'acc_test', 'f_train', 'f_test']
idx_learners = ['LogisticRegression', 'AdaBoostClassifier', 'GradientBoostingClassifie
r']
ranking = pd.DataFrame(columns= idx_metrics, index= idx_learners)

for i in range(len(idx_learners)):
    ranking.loc[idx_learners[i]] = pd.Series(results[idx_learners[i]][2])

pd.DataFrame(ranking)
```

Out[12]:

	train_time	pred_time	acc_train	acc_test	f_train	f
LogisticRegression	1.17584	0.0156238	0.843333	0.842012	0.690299	0.68
AdaBoostClassifier	2.25454	0.0937185	0.85	0.857601	0.711538	0.72
GradientBoostingClassifier	10.2635	0.03125	0.856667	0.863018	0.734127	0.73

Questão 4 - Descrevendo o modelo em termos leigos

- Em um ou dois parágrafos, explique para a *CharityML*, em termos leigos, como o modelo final escolhido deveria funcionar. Garanta que você está descrevendo as principais vantagens do modelo, tais como o modo de treinar o modelo e como o modelo realiza a predição. Evite a utilização de jargões matemáticos avançados, como por exemplo a descrição de equações.

DICA:

Quando estiver explicando seu modelo, cite as fontes externas utilizadas, caso utilize alguma.

Resposta:

Com o objetivo de classificar se a renda de um indivíduo é superior ou não a 50,000, o *Gradient Boosting Classifier* reúne sequencialmente diversos classificadores "fracos" que funcionam como uma árvore de decisão: usando as características presentes na base de dados como *age*, *occupation*, *capital-gain*, etc., esses classificadores criam regras simples baseadas em verificações de verdadeiro ou falso e valores de corte para classificar se um indivíduo é ou não um potencial doador à *CharityML*. Cada classificador decide individualmente se o candidato em questão tem ou não mais de 50,000 de renda e, então, os pareceres são reunidos e contabilizados como em um processo de votação ponderada, constituindo o que chamamos de "classificador forte".[7] (https://www.youtube.com/watch?v=ErDgauqnTHk)

Inicialmente, todos os indivíduos (observações) tem a mesma importância para o modelo e, como algoritmos de aprendizado se baseiam em tentativa e erro, nem todos os indivíduos são classificadas corretamente nesta primeira tentativa. Assim, durante as etapas de treinamento, o algoritmo verifica os indivíduos correta e erroneamente classificados quanto à renda e altera a importância de cada um, de modo a dar mais peso aos erroneamente classificados e menos peso aos corretamente classificados. O objetivo disto é ajustar as diferenças (ou resíduos) de classificação (*"gradiente"*) e priorizar a predição correta (*"boost"*). Novas iterações são feitas com as etapas de classificações independentes, contabilização das classificações e ajuste de pesos dos indivíduos por um número previamente especificado de vezes ou até o algoritmo não conseguir melhorar mais suas classificações. Uma vez treinado, podemos utilizar o *Gradient Boosting* - que, dentre os modelos testados, teve maior assertividade - para estimar as chances de um indivíduo doar à *CharityML* [8] (http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/)

Referências

[7] MALAKAR, Gopal. Introduction To Gradient Boosting algorithm (simplistic n graphical) - Machine Learning. 2018. Disponível em: https://www.youtube.com/watch?v=ErDgauqnTHk (https://www.youtube.com/watch?v=ErDgauqnTHk). Acesso em: 25 nov. 2018.

[8] A KAGGLE MASTER EXPLAINS GRADIENT BOOSTING. In: The Official Blog Of Kaggle. Disponível em: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/ (http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/). Acesso em: 25 nov. 2018

Implementação: Tuning do modelo

Refine o modelo escolhido. Utilize uma busca grid (`GridSearchCV`) com pelo menos um parâmetro importante refinado com pelo menos 3 valores diferentes. Você precisará utilizar todo o conjunto de treinamento para isso. Na célula de código abaixo, você precisará implementar o seguinte:

- Importar `sklearn.grid_search.GridSearchCV` (http://scikit-learn.org/0.17/modules/generated/sklearn.grid_search.GridSearchCV.html) e `sklearn.metrics.make_scorer` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- Inicializar o classificador escolhido por você e armazená-lo em `clf`.
  - Configurar um `random_state` se houver um disponível para o mesmo estado que você configurou anteriormente.
- Criar um dicionário dos parâmetros que você quer otimizar para o modelo escolhido.
  - Exemplo: `parâmetro = {'parâmetro' : [lista de valores]}`.
  - **Nota:** Evite otimizar o parâmetro `max_features` se este parâmetro estiver disponível!
- Utilize `make_scorer` para criar um objeto de pontuação `fbeta_score` (com $\beta = 0.5$).
- Realize a busca grid no classificador `clf` utilizando o `'scorer'` e armazene-o na variável `grid_obj`.
- Adeque o objeto da busca grid aos dados de treino (`X_train`, `y_train`) e armazene em `grid_fit`.

Nota: Dependendo do algoritmo escolhido e da lista de parâmetros, a implementação a seguir pode levar algum tempo para executar!

In [13]:

```
# TODO: Importar 'GridSearchCV', 'make_scorer', e qualquer biblioteca necessária
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

# TODO: Inicializar o classificador
clf = GradientBoostingClassifier(random_state= 0)

# TODO: Criar a lista de parâmetros que você quer otimizar, utilizando um dicionário, c
aso necessário.
# HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, value2]}

# Esta "Grid Search" tem levado, aproximadamente, 15 min para ser executada.
# Comentei o parâmetro de 'max_depth' pois, apesar de ser indicado para "tuning" do mod
elo,
# ele estava aumentando o tempo de treinamento por horas, sem melhorar em nada a acurác
ia e o f-score.

parameters = {'n_estimators' : [250, 500, 750],
              'learning_rate': [.5, .2, .1] #,
              #'max_depth'    : [3, 4, 5]
             }

# TODO: Criar um objeto fbeta_score utilizando make_scorer()
scorer = make_scorer(fbeta_score, beta= .5)

# TODO: Realizar uma busca grid no classificador utilizando o 'scorer' como o método de
 score no GridSearchCV()
grid_obj = GridSearchCV(clf,
                        param_grid= parameters,
                        scoring= scorer)

# TODO: Adequar o objeto da busca grid como os dados para treinamento e encontrar os pa
râmetros ótimos utilizando fit()
grid_fit = grid_obj.fit(X_train, y_train)

# Recuperar o estimador
best_clf = grid_fit.best_estimator_

# Realizar predições utilizando o modelo não otimizado e modelar
predictions = (clf.fit(X_train, y_train)).predict(X_test)
best_predictions = best_clf.predict(X_test)

# Reportar os scores de antes e de depois
print("Unoptimized model\n------")
print("Accuracy score on testing data: {:.4f}".format(accuracy_score(y_test, prediction
s)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, predictions, beta =
0.5)))
print("\nOptimized Model\n------")
print("Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test,
best_predictions)))
print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predi
ctions, beta = 0.5)))
```

```
Unoptimized model
------
Accuracy score on testing data: 0.8630
F-score on testing data: 0.7395

Optimized Model
------
Final accuracy score on the testing data: 0.8718
Final F-score on the testing data: 0.7545
```

Questão 5 - Validação final do modelo

- Qual é a *accuracy* e o F-score do modelo otimizado utilizando os dados de testes?
- Estes scores são melhores ou piores do que o modelo antes da otimização?
- Como os resultados do modelo otimizado se comparam aos *benchmarks* do *naive predictor* que você encontrou na **Questão 1**?

Nota: Preencha a tabela abaixo com seus resultados e então responda as questões no campo **Resposta**

Resultados:

Metric	Benchmark Model	Unoptimized Model	Optimized Model
Accuracy Score	0.2478	0.8630	0.8718
F-score	0.2917	0.7395	0.7545

Resposta:

- Sobre os dados de teste, a *accuracy* e o *F-score* do modelo otimizado são 0.8718 e 0.7545 respectivamente.
- Conforme esperado, estes *scores* são melhores do que os do modelo antes da otimização. Entretanto, a busca no *grid* demandou muito mais tempo e resultou em incrementos pouco expressivos: 0.009 na *accuracy* e 0.015 no *F-score*.
- Tanto a *accuracy* quanto o *F-score* do modelo otimizado são consideravelmente melhores do que os do *Naive Predictor*, existindo respectivos saltos de 0.6240 e 0.4628 entre o *Gradient Boosting* escolhido e o modelo *naive* de *benchmark*.

Importância dos atributos

Uma tarefa importante quando realizamos aprendizado supervisionado em um conjunto de dados como os dados do censo que estudamos aqui é determinar quais atributos fornecem maior poder de predição. Focar no relacionamento entre alguns poucos atributos mais importantes e na label alvo, nós simplificamos muito o nosso entendimento do fenômeno, que é a coisa mais importante a se fazer. No caso deste projeto, isso significa que nós queremos identificar um pequeno número de atributos que possuem maior chance de predizer se um indivíduo possui renda anual superior à $50,000.

Escolha um classificador da scikit-learn (e.x.: adaboost, random forests) que possua o atributo `feature_importances_`, que é uma função que calcula o ranking de importância dos atributos de acordo com o classificador escolhido. Na próxima célula python, ajuste este classificador para o conjunto de treinamento e utilize este atributo para determinar os 5 atributos mais importantes do conjunto de dados do censo.

Questão 6 - Observação da Relevância dos Atributos

Quando **Exploramos os dados**, vimos que existem treze atributos disponíveis para cada registro nos dados do censo. Destes treze atributos, quais os 5 atributos que você acredita que são os mais importantes para predição e em que ordem você os ranquearia? Por quê?

Resposta:

Considerando que desejamos fazer predições sobre a *renda* (<= 50K ou >50K), a meu ver, os 5 atributos mais relevantes, em ordem de importância, seriam:

1. *capital-gain*: deve haver uma forte correlação positiva entre os ganhos de capital e a renda de um candidato à doação;
2. *capital-loss*: da mesma forma, deve haver uma correlação negativa entre perdas de capital e renda; a diferença entre este atributo e o anterior - ou capital líquido - deve apresentar, de forma direta, uma forte correlação positiva com a renda;
3. *occupation*: parece razoável assumir que tipo de emprego e campo de trabalho refletem na renda pessoal - um *executivo* tende a ter mais renda que um *operário*, por exemplo;
4. *education-num*: ao indicar os anos de instrução, esta característica pode se correlacionar positivamente com a renda - quanto mais alto o nível educacional de um indivíduo, maior sua renda esperada.
5. *age*: em se tratando de acúmulo de patrimônio, espera-se que pessoas mais vividas e estabilizadas tenham mais renda do que as mais jovens e em início de carreira.

Implementação - Extraindo a importância do atributo

Escolha um algoritmo de aprendizado supervisionado da `scikit-learn` que possui o atributo `features_importance_` disponível. Este atributo é uma função que ranqueia a importância de cada atributo dos registros do conjunto de dados quando realizamos predições baseadas no algoritmo escolhido.

Na célula de código abaixo, você precisará implementar o seguinte:

- Importar um modelo de aprendizado supervisionado da sklearn se este for diferente dos três usados anteriormente.
- Treinar o modelo supervisionado com todo o conjunto de treinamento.
- Extrair a importância dos atributos utilizando `'.feature_importances_'`.

In [14]:

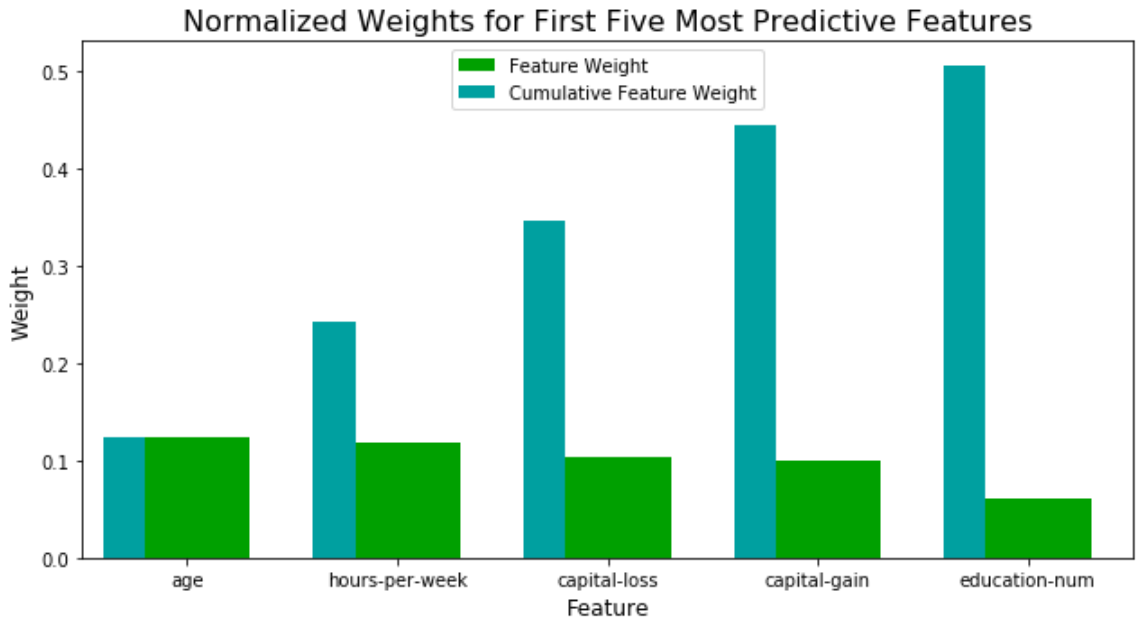
```
# TODO: Importar um modelo de aprendizado supervisionado que tenha 'feature_importances
_'
from sklearn.ensemble import AdaBoostClassifier

# TODO: Treinar o modelo utilizando o conjunto de treinamento com .fit(X_train, y_trai
n)

model = GradientBoostingClassifier(random_state= 0,
                                   n_estimators= 750).fit(X_train, y_train)


# TODO: Extrair a importância dos atributos utilizando .feature_importances_
importances = model.feature_importances_

# Plotar
vs.feature_plot(importances, X_train, y_train)
```



Questão 7 - Extraindo importância dos atributos

Observe a visualização criada acima que exibe os cinco atributos mais relevantes para predizer se um indivíduo possui remuneração igual ou superior à $50,000 por ano.

- Como estes cinco atributos se comparam com os 5 atributos que você discutiu na **Questão 6**?
- Se você estivesse próximo da mesma resposta, como esta visualização confirma o seu raciocínio?
- Se você não estava próximo, por que você acha que estes atributos são mais relevantes?

Resposta:

- 4 dos 5 atributos apontados como relevantes foram confirmados como importantes pelo *Gradiente Boosting*: *age*, *capital-gain*, *capital-loss* e *education-num*. A ordem de importância, porém, apresentou divergências: enquanto eu ranquei *age* com a menor importância, o classificador considerou este o atributo mais importante. Entretanto, interpretando-se o gráfico de barras *"Normalized Weights For Five Most More Predictive Features"*, como os valores de importância de pares de atributos como *age* e *capital-loss* e *capital-gain* e *education-num* são muito próximos entre si, eles não invalidam totalmente meu *rank* inicial (vide tabela abaixo). Um atributo inesperado indicado como relevante foi *hours-per-week* em vez de *occupation*, mas, não tão inesperado assim, porque é factível se supor uma correlação positiva entre profissão, quantidade de horas trabalhadas por semana e renda.
- Como, ao menos no aspecto qualitativo, as listas de atributos foram bem (80%) similares, pode-se perceber uma coerência entre minha intuição para a hipótese inicial e a confirmação através da importância matematicamente computada.

Comparativo de Importância de Atributos

Rank	Apontado	Verificado	Importância
1	*capital-gain*	*age*	0.1247
2	*capital-loss*	*hours-per-week*	0.1189
3	*occupation*	*capital-loss*	0.1033
4	*education-num*	*capital-gain*	0.0990
5	*age*	*education-num*	0.0603

Selecionando atributos

Como um modelo performa se nós só utilizamos um subconjunto de todos os atributos disponíveis nos dados? Com menos atributos necessários para treinar, a expectativa é que o treinamento e a predição sejam executados em um tempo muito menor — com o custo da redução nas métricas de performance. A partir da visualização acima, nós vemos que os cinco atributos mais importantes contribuem para mais de 50% da importância de **todos** os atributos presentes nos dados. Isto indica que nós podemos tentar *reduzir os atributos* e simplificar a informação necessária para o modelo aprender. O código abaixo utilizará o mesmo modelo otimizado que você encontrou anteriormente e treinará o modelo com o mesmo conjunto de dados de treinamento, porém apenas com *os cinco atributos mais importantes*

In [15]:

```
# Importar a funcionalidade para clonar um modelo
from sklearn.base import clone

# Reduzir a quantidade de atributos
X_train_reduced = X_train[X_train.columns.values[(np.argsort(importances)[::-1])[:5]]]
X_test_reduced = X_test[X_test.columns.values[(np.argsort(importances)[::-1])[:5]]]

# Treinar o melhor modelo encontrado com a busca grid anterior
clf = (clone(best_clf)).fit(X_train_reduced, y_train)

# Fazer novas predições
reduced_predictions = clf.predict(X_test_reduced)

# Reportar os scores do modelo final utilizando as duas versões dos dados.
print("Final Model trained on full data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, best_predictions
)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, be
ta = 0.5)))
print("\nFinal Model trained on reduced data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduced_predicti
ons)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_predictions,
 beta = 0.5)))
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8718
F-score on testing data: 0.7545

Final Model trained on reduced data
------
Accuracy on testing data: 0.8425
F-score on testing data: 0.6993
```

Questão 8 - Efeitos da seleção de atributos

- Como o F-score do modelo final e o accuracy score do conjunto de dados reduzido utilizando apenas cinco atributos se compara aos mesmos indicadores utilizando todos os atributos?
- Se o tempo de treinamento é uma variável importante, você consideraria utilizar os dados enxutos como seu conjunto de treinamento?

Resposta:

- O *F-score* e a *accuracy* do modelo final no conjunto de dados reduzidos são menores do que o modelo aplicado a todos os atributos. A *accuracy* teve uma redução de 0.0293, caindo de 0.8718 para 0.8425 e o *F-score* teve uma redução de 0.0552, caindo de 0.7545 para 0.6993.
- Se o tempo de treinamento fosse uma variável importante, eu treinaria o modelo com base nos dados enxutos. Entretanto, como aplicação do modelo é de treinamento periódico e, ao ser treinado com o conjunto enxuto de dados, o *F-score* do *Gradient Boosting* apresentou uma redução de 5.5% - inclusive bem superior ao ganho de 1.5% com a otimização através da *Grid Search* -, eu sugeriria trabalhar como o conjunto de dados completo.

Comparativo entre Modelos:

Métrica	Modelo Inicial	Modelo Otimizado	Modelo Reduzido
Accuracy Score	0.8630	0.8718	0.8425
F-score	0.7395	0.7545	0.6993

Nota: Uma vez que você tenha concluído toda a implementação de código e respondido cada uma das questões acima, você poderá finalizar o seu trabalho exportando o iPython Notebook como um documento HTML. Você pode fazer isso utilizando o menu acima navegando para **File -> Download as -> HTML (.html)**. Inclua este documento junto do seu notebook como sua submissão.

Testes Extras:

Escolhendo os melhores *learners* candidatos ao problema no `sklearn`

In [16]:

```
from sklearn.linear_model import LogisticRegression, Perceptron, SGDClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, ExtraTreesClas
sifier, GradientBoostingClassifier, BaggingClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

# Número de amostras:
n_samp = len(income)
#n_samp = 5000

# Classe de modelos lineares generalizados
mod1 = train_predict(LogisticRegression(solver= 'lbfgs', max_iter= 500), n_samp, X_trai
n, y_train, X_test, y_test)
mod2 = train_predict(Perceptron(max_iter = 500), n_samp, X_train, y_train, X_test, y_te
st)
mod3 = train_predict(SGDClassifier(random_state= 0, max_iter= 500), n_samp, X_train, y_
train, X_test, y_test)

# Classe de modelos Gaussianos
mod4 = train_predict(GaussianNB(), n_samp, X_train, y_train, X_test, y_test)

# Classe de modelos Ensemble
mod5 = train_predict(RandomForestClassifier(n_estimators= 500), n_samp, X_train, y_trai
n, X_test, y_test)
mod6 = train_predict(AdaBoostClassifier(random_state= 0, n_estimators= 500), n_samp, X_
train, y_train, X_test, y_test)
mod7 = train_predict(ExtraTreesClassifier(random_state= 0, n_estimators= 500), n_samp,
X_train, y_train, X_test, y_test)
mod8 = train_predict(GradientBoostingClassifier(random_state= 0, n_estimators= 500), n_
samp, X_train, y_train, X_test, y_test)
mod9 = train_predict(BaggingClassifier(random_state= 0, n_estimators= 500), n_samp, X_t
rain, y_train, X_test, y_test)

# Classe de modelos SVM
mod10 = train_predict(SVC(random_state= 0), n_samp, X_train, y_train, X_test, y_test)

# Classe de modelos Neighbors
mod11 = train_predict(KNeighborsClassifier(), n_samp, X_train, y_train, X_test, y_test)

# Classe de modelos Tree
mod12 = train_predict(DecisionTreeClassifier(random_state= 0), n_samp, X_train, y_train
, X_test, y_test)
```

```
LogisticRegression trained on 45222 samples.
Perceptron trained on 45222 samples.
SGDClassifier trained on 45222 samples.
GaussianNB trained on 45222 samples.
RandomForestClassifier trained on 45222 samples.
AdaBoostClassifier trained on 45222 samples.
ExtraTreesClassifier trained on 45222 samples.
GradientBoostingClassifier trained on 45222 samples.
BaggingClassifier trained on 45222 samples.
SVC trained on 45222 samples.
KNeighborsClassifier trained on 45222 samples.
DecisionTreeClassifier trained on 45222 samples.
```

In [17]:

```
# Dataframe de benchmarking para pré-seleção

idx_metrics = ['train_time', 'pred_time', 'acc_train', 'acc_test', 'f_train', 'f_test']

idx_all_learners = ['LogisticRegression', 'Perceptron', 'StochasticGradient', 'Gaussian
NB','RandomForest','AdaBoost',
                'ExtraTrees', 'GradientBoosting', 'Bagging', 'SVC', 'KNeighbors', 'Deci
sionTree']

ranking = pd.DataFrame(columns= idx_metrics, index= idx_all_learners)

for i in range(len(idx_all_learners)):
    ranking.loc[idx_all_learners[i]] = pd.Series(globals()['mod{}'.format(i+1)])

sorted_rank = ranking.sort_values(by= ['f_test', 'acc_test', 'train_time'],
                                  ascending= [False, False, True])
display(sorted_rank)
```

	train_time	pred_time	acc_train	acc_test	f_train	f_test
GradientBoosting	47.9135	0.108938	0.883333	0.871752	0.792254	0.754539
AdaBoost	21.0699	0.898486	0.863333	0.866446	0.746269	0.743236
Bagging	179.796	7.15975	0.976667	0.847319	0.970588	0.691557
RandomForest	36.8469	1.70802	0.976667	0.844334	0.970588	0.685655
LogisticRegression	3.19317	0.0129943	0.846667	0.841791	0.698529	0.682929
StochasticGradient	8.67103	0.0129929	0.853333	0.838695	0.719697	0.676636
SVC	106.836	18.711	0.853333	0.837148	0.719697	0.674477
ExtraTrees	55.3802	2.46259	0.97	0.824544	0.963855	0.641175
KNeighbors	1.45717	29.8499	0.883333	0.823549	0.771605	0.639164
DecisionTree	0.78255	0.0149918	0.97	0.818574	0.963855	0.627939
Perceptron	8.66603	0.0149918	0.816667	0.791819	0.617089	0.573185
GaussianNB	0.301827	0.0539687	0.593333	0.597678	0.4125	0.420899