Machine Learning Engineer Nanodegree

Supervised Learning

Project: Finding Donors for *CharityML*

Welcome to the second project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been provided for you, and it will be your job to implement the additional functionality necessary to successfully complete this project. Sections that begin with **'Implementation'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a `'TODO'` statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Please specify WHICH VERSION OF PYTHON you are using when submitting this notebook. Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Getting Started

In this project, you will employ several supervised algorithms of your choice to accurately model individuals' income using data collected from the 1994 U.S. Census. You will then choose the best candidate algorithm from preliminary results and further optimize this algorithm to best model the data. Your goal with this implementation is to construct a model that accurately predicts whether an individual makes more than $50,000. This sort of task can arise in a non-profit setting, where organizations survive on donations. Understanding an individual's income can help a non-profit better understand how large of a donation to request, or whether or not they should reach out to begin with. While it can be difficult to determine an individual's general income bracket directly from public sources, we can (as we will see) infer this value from other publically available features.

The dataset for this project originates from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Census+Income). The datset was donated by Ron Kohavi and Barry Becker, after being published in the article "*Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid*". You can find the article by Ron Kohavi online (https://www.aaai.org/Papers/KDD/1996/KDD96-033.pdf). The data we investigate here consists of small changes to the original dataset, such as removing the `'fnlwgt'` feature and records with missing or ill-formatted entries.

Exploring the Data

Run the code cell below to load necessary Python libraries and load the census data. Note that the last column from this dataset, `'income'`, will be our target label (whether an individual makes more than, or at most, $50,000 annually). All other columns are features about each individual in the census database.

In [1]:
```
# Import libraries necessary for this project
import numpy as np
import pandas as pd
from time import time
from IPython.display import display # Allows the use of display() for DataFrames

# Import supplementary visualization code visuals.py
import visuals as vs

# Pretty display for notebooks
%matplotlib inline

# Load the Census dataset
data = pd.read_csv("census.csv")

# Success - Display the first record
display(data.head(n=1))
```

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

Implementation: Data Exploration

A cursory investigation of the dataset will determine how many individuals fit into either group, and will tell us about the percentage of these individuals making more than $50,000. In the code cell below, you will need to compute the following:

- The total number of records, `'n_records'`
- The number of individuals making more than $50,000 annually, `'n_greater_50k'`.
- The number of individuals making at most $50,000 annually, `'n_at_most_50k'`.
- The percentage of individuals making more than $50,000 annually, `'greater_percent'`.

Hint: You may need to look at the table above to understand how the `'income'` entries are formatted.

In [8]:
```
# TODO: Total number of records
n_records = data.shape[0]

# TODO: Number of records where individual's income is more than $50,000
greater_50k = data.loc[lambda record: record.income == '>50K', :]
n_greater_50k = greater_50k.size

# TODO: Number of records where individual's income is at most $50,000
at_most_50k = data.loc[lambda record: record.income == '<=50K', :]
n_at_most_50k = at_most_50k.size

# TODO: Percentage of individuals whose income is more than $50,000
greater_percent = ((n_greater_50k * 1.0) / (n_greater_50k * 1.0 + n_at_m
ost_50k * 1.0)) * 100.0

# Print the results
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}%".for
mat(greater_percent))
```

```
Total number of records: 45222
Individuals making more than $50,000: 156912
Individuals making at most $50,000: 476196
Percentage of individuals making more than $50,000: 24.78%
```

Preparing the Data

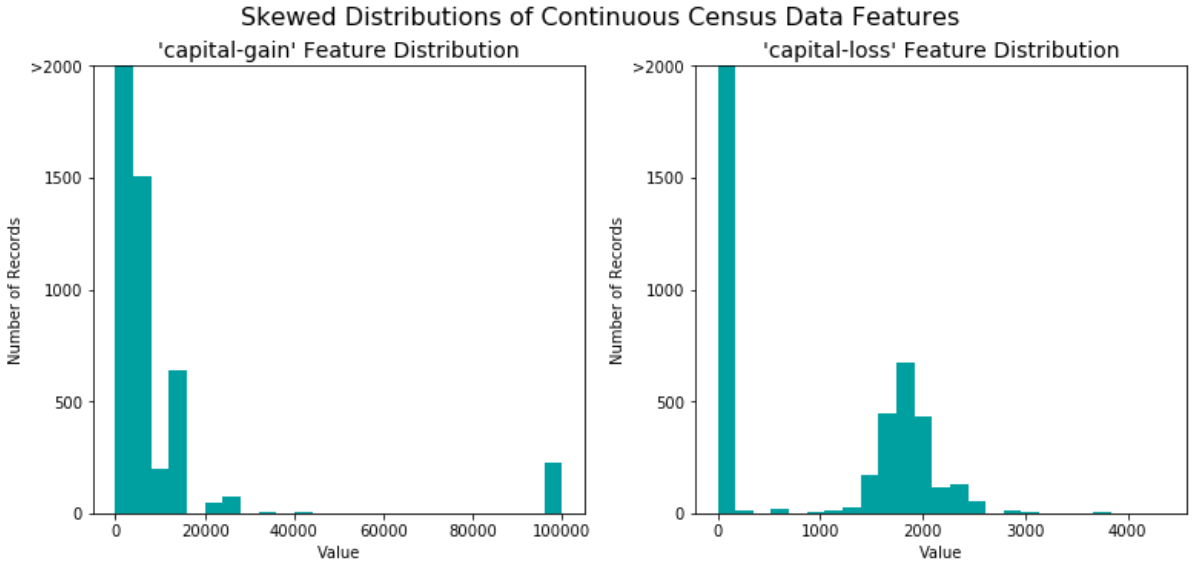
Before data can be used as input for machine learning algorithms, it often must be cleaned, formatted, and restructured — this is typically known as **preprocessing**. Fortunately, for this dataset, there are no invalid or missing entries we must deal with, however, there are some qualities about certain features that must be adjusted. This preprocessing can help tremendously with the outcome and predictive power of nearly all learning algorithms.

Transforming Skewed Continuous Features

A dataset may sometimes contain at least one feature whose values tend to lie near a single number, but will also have a non-trivial number of vastly larger or smaller values than that single number. Algorithms can be sensitive to such distributions of values and can underperform if the range is not properly normalized. With the census dataset two features fit this description: `'capital-gain'` and `'capital-loss'`.

Run the code cell below to plot a histogram of these two features. Note the range of the values present and how they are distributed.

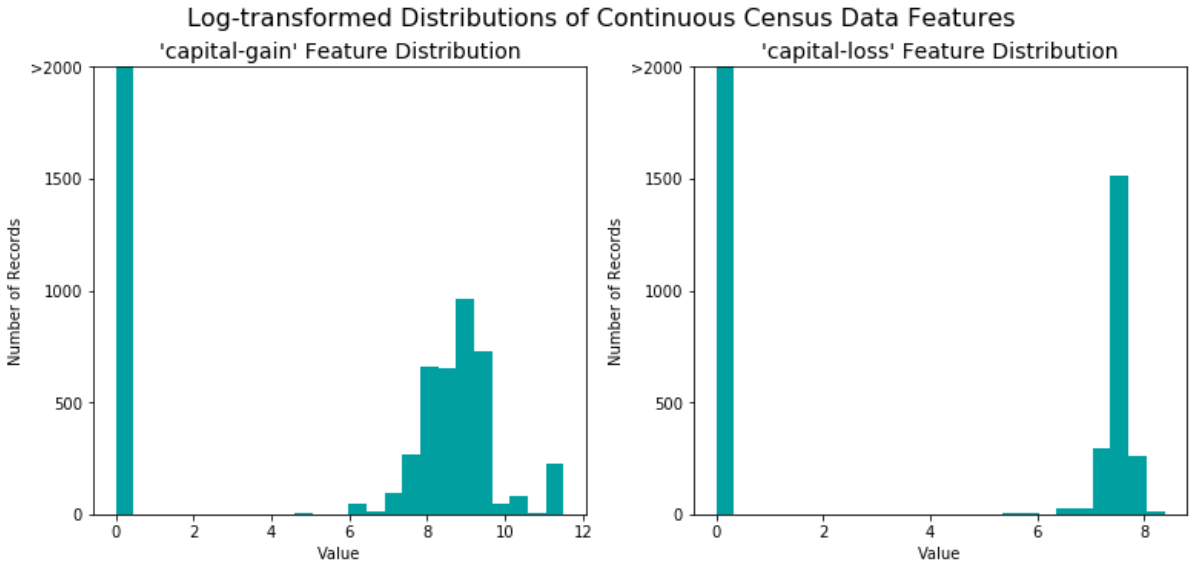
In [9]:
```
# Split the data into features and target label
income_raw = data['income']
features_raw = data.drop('income', axis = 1)

# Visualize skewed continuous features of original data
vs.distribution(data)
```



For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'`, it is common practice to apply a logarithmic transformation (https://en.wikipedia.org/wiki/Data_transformation_(statistics)) on the data so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation significantly reduces the range of values caused by outliers. Care must be taken when applying this transformation however: The logarithm of `0` is undefined, so we must translate the values by a small amount above `0` to apply the the logarithm successfully.

Run the code cell below to perform a transformation on the data and visualize the results. Again, note the range of values and how they are distributed.

In [10]:
```
# Log-transform the skewed features
skewed = ['capital-gain', 'capital-loss']
features_raw[skewed] = data[skewed].apply(lambda x: np.log(x + 1))

# Visualize the new log distributions
vs.distribution(features_raw, transformed = True)
```



Normalizing Numerical Features

In addition to performing transformations on features that are highly skewed, it is often good practice to perform some type of scaling on numerical features. Applying a scaling to the data does not change the shape of each feature's distribution (such as `'capital-gain'` or `'capital-loss'` above); however, normalization ensures that each feature is treated equally when applying supervised learners. Note that once scaling is applied, observing the data in its raw form will no longer have the same original meaning, as exampled below.

Run the code cell below to normalize each numerical feature. We will use `sklearn.preprocessing.MinMaxScaler` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html) for this.

In [11]:
```
# Import sklearn.preprocessing.StandardScaler
from sklearn.preprocessing import MinMaxScaler

# Initialize a scaler, then apply it to the features
scaler = MinMaxScaler()
numerical = ['age', 'education-num', 'capital-gain', 'capital-loss', 'ho
urs-per-week']
features_raw[numerical] = scaler.fit_transform(features_raw[numerical])

# Show an example of a record with scaling applied
display(features_raw.head(n = 1))
```

	age	workclass	education_level	education-num	marital-status	occupation	relationship	rac
0	0.30137	State-gov	Bachelors	0.8	Never-married	Adm-clerical	Not-in-family	Wh

Implementation: Data Preprocessing

From the table in **Exploring the Data** above, we can see there are several features for each record that are non-numeric. Typically, learning algorithms expect input to be numeric, which requires that non-numeric features (called *categorical variables*) be converted. One popular way to convert categorical variables is by using the **one-hot encoding** scheme. One-hot encoding creates a "*dummy*" variable for each possible category of each non-numeric feature. For example, assume `someFeature` has three possible entries: `A`, `B`, or `C`. We then encode this feature into `someFeature_A`, `someFeature_B` and `someFeature_C`.

	someFeature		someFeature_A	someFeature_B	someFeature_C
0	B		0	1	0
1	C	----> one-hot encode ---->	0	0	1
2	A		1	0	0

Additionally, as with the non-numeric features, we need to convert the non-numeric target label, `'income'` to numerical values for the learning algorithm to work. Since there are only two possible categories for this label ("<=50K" and ">50K"), we can avoid using one-hot encoding and simply encode these two categories as `0` and `1`, respectively. In code cell below, you will need to implement the following:

- Use `pandas.get_dummies()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html?highlight=get_dummies#pandas.get_dummies) to perform one-hot encoding on the `'features_raw'` data.
- Convert the target label `'income_raw'` to numerical entries.
  - Set records with "<=50K" to `0` and records with ">50K" to `1`.

In [14]:
```
# TODO: One-hot encode the 'features_raw' data using pandas.get_dummies
()
features = pd.get_dummies(features_raw)

# TODO: Encode the 'income_raw' data to numerical values
income = [0 if record == '<=50K' else 1 for record in income_raw]

# Print the number of features after one-hot encoding
encoded = list(features.columns)
print ("{} total features after one-hot encoding.".format(len(encoded)))

# Uncomment the following line to see the encoded feature names
print (encoded)
```

```
103 total features after one-hot encoding.
['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-wee
k', 'workclass_ Federal-gov', 'workclass_ Local-gov', 'workclass_ Priva
te', 'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc', 'workcla
ss_ State-gov', 'workclass_ Without-pay', 'education_level_ 10th', 'edu
cation_level_ 11th', 'education_level_ 12th', 'education_level_ 1st-4t
h', 'education_level_ 5th-6th', 'education_level_ 7th-8th', 'education_
level_ 9th', 'education_level_ Assoc-acdm', 'education_level_ Assoc-vo
c', 'education_level_ Bachelors', 'education_level_ Doctorate', 'educat
ion_level_ HS-grad', 'education_level_ Masters', 'education_level_ Pres
chool', 'education_level_ Prof-school', 'education_level_ Some-colleg
e', 'marital-status_ Divorced', 'marital-status_ Married-AF-spouse', 'm
arital-status_ Married-civ-spouse', 'marital-status_ Married-spouse-abs
ent', 'marital-status_ Never-married', 'marital-status_ Separated', 'ma
rital-status_ Widowed', 'occupation_ Adm-clerical', 'occupation_ Armed-
Forces', 'occupation_ Craft-repair', 'occupation_ Exec-managerial', 'oc
cupation_ Farming-fishing', 'occupation_ Handlers-cleaners', 'occupatio
n_ Machine-op-inspct', 'occupation_ Other-service', 'occupation_ Priv-h
ouse-serv', 'occupation_ Prof-specialty', 'occupation_ Protective-ser
v', 'occupation_ Sales', 'occupation_ Tech-support', 'occupation_ Trans
port-moving', 'relationship_ Husband', 'relationship_ Not-in-family',
'relationship_ Other-relative', 'relationship_ Own-child', 'relationshi
p_ Unmarried', 'relationship_ Wife', 'race_ Amer-Indian-Eskimo', 'race_
Asian-Pac-Islander', 'race_ Black', 'race_ Other', 'race_ White', 'sex_
Female', 'sex_ Male', 'native-country_ Cambodia', 'native-country_ Cana
da', 'native-country_ China', 'native-country_ Columbia', 'native-count
ry_ Cuba', 'native-country_ Dominican-Republic', 'native-country_ Ecuad
or', 'native-country_ El-Salvador', 'native-country_ England', 'native-
country_ France', 'native-country_ Germany', 'native-country_ Greece',
'native-country_ Guatemala', 'native-country_ Haiti', 'native-country_
Holand-Netherlands', 'native-country_ Honduras', 'native-country_ Hon
g', 'native-country_ Hungary', 'native-country_ India', 'native-country
_ Iran', 'native-country_ Ireland', 'native-country_ Italy', 'native-co
untry_ Jamaica', 'native-country_ Japan', 'native-country_ Laos', 'nati
ve-country_ Mexico', 'native-country_ Nicaragua', 'native-country_ Outl
ying-US(Guam-USVI-etc)', 'native-country_ Peru', 'native-country_ Phili
ppines', 'native-country_ Poland', 'native-country_ Portugal', 'native-
country_ Puerto-Rico', 'native-country_ Scotland', 'native-country_ Sou
th', 'native-country_ Taiwan', 'native-country_ Thailand', 'native-coun
try_ Trinadad&Tobago', 'native-country_ United-States', 'native-country
_ Vietnam', 'native-country_ Yugoslavia']
```

Shuffle and Split Data

Now all *categorical variables* have been converted into numerical features, and all numerical features have been normalized. As always, we will now split the data (both features and their labels) into training and test sets. 80% of the data will be used for training and 20% for testing.

Run the code cell below to perform this split.

In [15]:
```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Split the 'features' and 'income' data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, income, te
st_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.
```

Evaluating Model Performance

In this section, we will investigate four different algorithms, and determine which is best at modeling the data. Three of these algorithms will be supervised learners of your choice, and the fourth algorithm is known as a *naive predictor*.

Metrics and the Naive Predictor

CharityML, equipped with their research, knows individuals that make more than $50,000 are most likely to donate to their charity. Because of this, *CharityML* is particularly interested in predicting who makes more than $50,000 accurately. It would seem that using **accuracy** as a metric for evaluating a particular model's performace would be appropriate. Additionally, identifying someone that *does not* make more than $50,000 as someone who does would be detrimental to *CharityML*, since they are looking to find individuals willing to donate. Therefore, a model's ability to precisely predict those that make more than $50,000 is *more important* than the model's ability to **recall** those individuals. We can use **F-beta score** as a metric that considers both precision and recall:

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

In particular, when $\beta = 0.5$, more emphasis is placed on precision. This is called the **$F_{0.5}$ score** (or F-score for simplicity).

Looking at the distribution of classes (those who make at most $50,000, and those who make more), it's clear most individuals do not make more than $50,000. This can greatly affect **accuracy**, since we could simply say "*this person does not make more than $50,000*" and generally be right, without ever looking at the data! Making such a statement would be called **naive**, since we have not considered any information to substantiate the claim. It is always important to consider the *naive prediction* for your data, to help establish a benchmark for whether a model is performing well. That been said, using that prediction would be pointless: If we predicted all people made less than $50,000, *CharityML* would identify no one as donors.

Question 1 - Naive Predictor Performace

If we chose a model that always predicted an individual made more than $50,000, what would that model's accuracy and F-score be on this dataset?
Note: You must use the code cell below and assign your results to `'accuracy'` and `'fscore'` to be used later.

In [16]:
```
# TODO: Calculate accuracy
accuracy = (n_greater_50k * 1.0) / (n_greater_50k * 1.0 + n_at_most_50k
* 1.0)

# TODO: Calculate F-score using the formula above for beta = 0.5
tp = accuracy
fp = (n_at_most_50k * 1.0) / (n_greater_50k * 1.0 + n_at_most_50k * 1.0)
fn = 0.0
precision = tp / (tp + fp)
recall = tp / (tp + fn)
beta = 0.5
fscore = (1.0 + beta * beta) * ((precision * recall) / (beta * beta * pr
ecision + recall))

# Print the results
print ("Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".form
at(accuracy, fscore))
```

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

Supervised Learning Models

The following supervised learning models are currently available in `scikit-learn` (http://scikit-learn.org/stable/supervised_learning.html) **that you may choose from:**

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

Question 2 - Model Application

List three of the supervised learning models above that are appropriate for this problem that you will test on the census data. For each model chosen

- *Describe one real-world application in industry where the model can be applied.* (You may need to do research for this — give references!)
- *What are the strengths of the model; when does it perform well?*
- *What are the weaknesses of the model; when does it perform poorly?*
- *What makes this model a good candidate for the problem, given what you know about the data?*

Answer:

Decision Tree

- Decision tree has been widely used for classification problems (see http://www.cbcb.umd.edu/~salzberg/docs/murthy_thesis/survey/node32.html (http://www.cbcb.umd.edu/~salzberg/docs/murthy_thesis/survey/node32.html))

In CapitalOne, we used decision tree for the probablistic match of customer records.

- The strengs of the model include: easy interperation, fast, implicit feature selection, relatively little data preprocessing (e.g., no need for scaling/normalization, outliers remove, missing data), capability of handling non-linear relationships, ..., etc.

Decision tree performs well when the data can be splitted into different regions along dimension lines (e.g., horizontally and vertically in two dimensional space).

- The weaknesses of the model include: over-fitting, difficulity in handling data region boundaries that are not parallel to dimension lines (e.g., a diagonal line).
- According to the label of the dataset, this project is a classification problem and the data pattern is not linear. Decision tree is good at this kind of problem in general.

Random Forest

- Again, in CapitalOne, we used Random Forest for the probabilistic match of customer records.
- The strengs of the model include: As an ensemble method of decision trees, it inherits most of the advantages of Decision Tree. In addition, it has many advantages compared with Decision Tree, including: significantly better prediction accuracy, more resilient to over-fitting, ..., etc.

Like Decision Tree, the model performs well when the data can be splitted into different regions along dimension lines (e.g., horizontally and vertically in two dimensional space).

- The weaknesses of the model include: no intuitive interperation, over-fitting, difficulity in handling data region boundaries that are not parallel to dimension lines (e.g., a diagonal line).
- According to the label of the dataset, this project is a classification problem and the data pattern is not linear. Random Forest is good fit to this kind of problem in general.

Gradient Boosting

- Again, in CapitalOne, we also used Gradient Boosting for the probabilistic match of customer records.
- The strengs of the model include: As an ensemble method of decision trees, it inherits most of the advantages of Decision Tree. In addition, it has advantages compared with Decision Tree, including: significantly better prediction accuracy, more resilient to over-fitting, ..., etc. The model tends to have competitive performance in accuracy and much less prediction time compared with Random Forest.

Like Decision Tree, the model performs well when the data can be splitted into different regions along dimension lines (e.g., horizontally and vertically in two dimensional space).

- The weaknesses of the model include: no intuitive interperation, over-fitting, difficulity in handling data region boundaries that are not parallel to dimension lines (e.g., a diagonal line). It is much slower than Random Forest in model training.

- According to the label of the dataset, this project is a classification problem and the data pattern is not linear. Like Random Forest, the Gradient Boosting is good fit to this kind of problem in general. I select Gradient Boosting over Random Forest is because the model is much fast in prediction.

Implementation - Creating a Training and Predicting Pipeline

To properly evaluate the performance of each model you've chosen, it's important that you create a training and predicting pipeline that allows you to quickly and effectively train models using various sizes of training data and perform predictions on the testing data. Your implementation here will be used in the following section. In the code block below, you will need to implement the following:

- Import `fbeta_score` and `accuracy_score` from `sklearn.metrics` (http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics).
- Fit the learner to the sampled training data and record the training time.
- Perform predictions on the test data `X_test`, and also on the first 300 training points `X_train[:300]`.
  - Record the total prediction time.
- Calculate the accuracy score for both the training subset and testing set.
- Calculate the F-score for both the training subset and testing set.
  - Make sure that you set the `beta` parameter!

In [17]:
```
# TODO: Import two metrics from sklearn - fbeta_score and accuracy_score
from sklearn.metrics import fbeta_score
from sklearn.metrics import 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 trai
ning 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 'sam
ple_size'
    start = time() # Get start time
    learner = 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,
    #       then get predictions on the first 300 training samples
    start = time() # Get start time

    predictions_test = learner.predict(X_test)
    # convert array to list for fbeta_score
    predictions_test = predictions_test.tolist()

    predictions_train = learner.predict(X_train[:300])
    # convert array to list for fbeta_score
    predictions_train = predictions_train.tolist()

    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
    results['acc_train'] = accuracy_score(y_train[:300], predictions_tra
in, normalize = True)

    # TODO: Compute accuracy on test set
    results['acc_test'] = accuracy_score(y_test, predictions_test, norma
lize = True)

    # TODO: Compute F-score on the the first 300 training samples
    results['f_train'] = fbeta_score(y_train[:300], predictions_train, a
verage='binary', beta=0.5)
```

```
    # TODO: Compute F-score on the test set
    results['f_test'] = fbeta_score(y_test, predictions_test, average='b
inary', beta=0.5)

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

    # Return the results
    return results
```

Implementation: Initial Model Evaluation

In the code cell, you will need to implement the following:

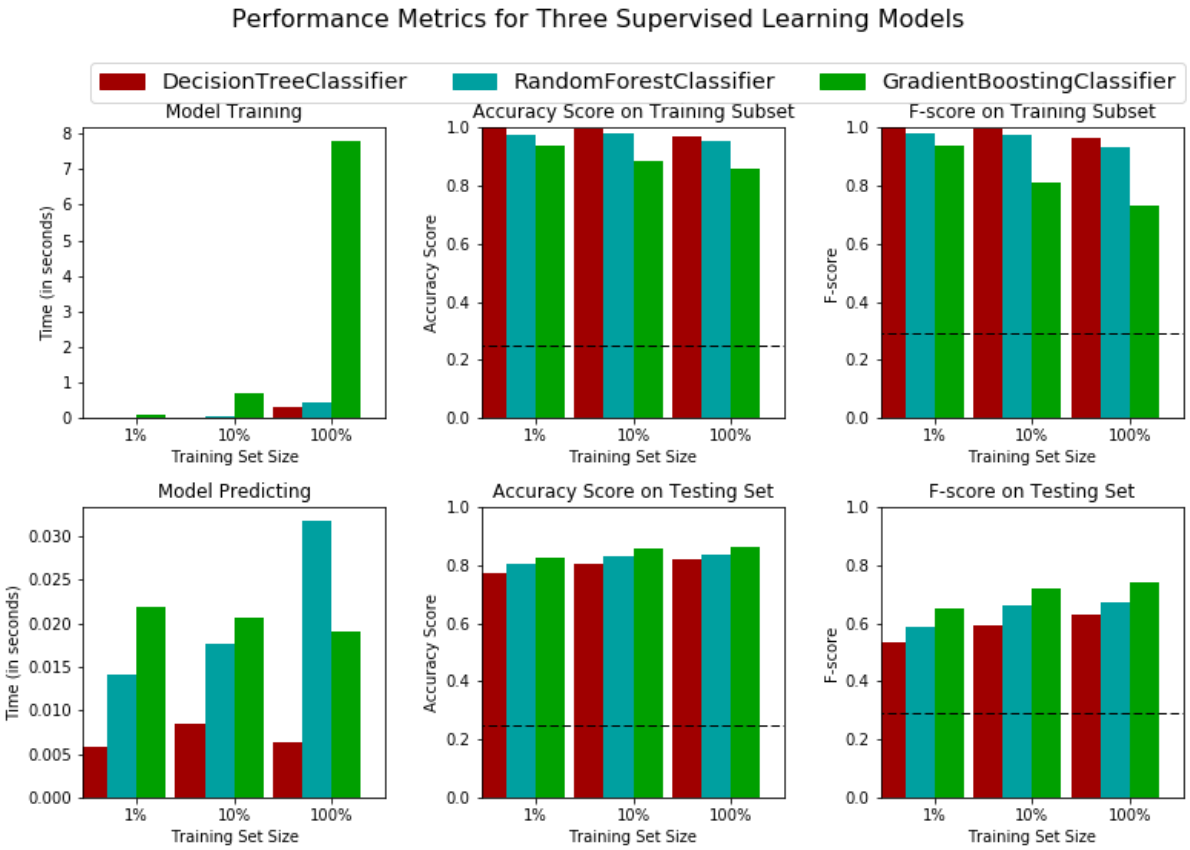
- Import the three supervised learning models you've discussed in the previous section.
- Initialize the three models and store them in `'clf_A'`, `'clf_B'`, and `'clf_C'`.
  - Use a `'random_state'` for each model you use, if provided.
  - **Note:** Use the default settings for each model — you will tune one specific model in a later section.
- Calculate the number of records equal to 1%, 10%, and 100% of the training data.
  - Store those values in `'samples_1'`, `'samples_10'`, and `'samples_100'` respectively.

Note: Depending on which algorithms you chose, the following implementation may take some time to run!

In [25]:
```
# TODO: Import the three supervised learning models from sklearn
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier

# TODO: Initialize the three models
clf_A = DecisionTreeClassifier(random_state=0)
clf_B = RandomForestClassifier(random_state=0)
clf_C = GradientBoostingClassifier(random_state=0)

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the tra
ining data
samples_1 = int(len(X_train) / 100)
samples_10 = int(len(X_train) / 10)
samples_100 = int(len(X_train))

# Collect results on the learners
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)

# Run metrics visualization for the three supervised learning models cho
sen
print("results: ")
print(results)

vs.evaluate(results, accuracy, fscore)
```

```
DecisionTreeClassifier trained on 361 samples.
DecisionTreeClassifier trained on 3617 samples.
DecisionTreeClassifier trained on 36177 samples.
RandomForestClassifier trained on 361 samples.
RandomForestClassifier trained on 3617 samples.
RandomForestClassifier trained on 36177 samples.
GradientBoostingClassifier trained on 361 samples.
GradientBoostingClassifier trained on 3617 samples.
GradientBoostingClassifier trained on 36177 samples.
results:
{'DecisionTreeClassifier': {0: {'train_time': 0.0022258758544921875, 'p
red_time': 0.0057239532470703125, 'acc_train': 1.0, 'acc_test': 0.77191
818684355995, 'f_train': 1.0, 'f_test': 0.5359784216479596l}, 1: {'trai
n_time': 0.023253917694091797, 'pred_time': 0.00849604606628418, 'acc_t
rain': 0.9966666666666667, 'acc_test': 0.80165837479270319, 'f_train':
0.99719101123595499, 'f_test': 0.5938748335552595}, 2: {'train_time':
0.32678985595703125, 'pred_time': 0.006402015686035156, 'acc_train': 0.
96999999999999997, 'acc_test': 0.81857379767827532, 'f_train': 0.963855
42168674709, 'f_test': 0.62793914246196403}}, 'RandomForestClassifier':
{0: {'train_time': 0.01236414909362793, 'pred_time': 0.014022827148437
5, 'acc_train': 0.97666666666666668, 'acc_test': 0.80254284134881149,
'f_train': 0.97891566265060259, 'f_test': 0.58670218927371709}, 1: {'tr
ain_time': 0.03684687614440918, 'pred_time': 0.01765918731689453, 'acc_
train': 0.97999999999999998, 'acc_test': 0.83228302929795472, 'f_trai
n': 0.97383720930232565, 'f_test': 0.66207569677909139}, 2: {'train_tim
e': 0.4586770534515381, 'pred_time': 0.03177928924560547, 'acc_train':
0.95666666666666667, 'acc_test': 0.83781094527363187, 'f_train': 0.9337
3493975903621, 'f_test': 0.67173891280408449}}, 'GradientBoostingClassi
fier': {0: {'train_time': 0.08863401412963867, 'pred_time': 0.021855115
89050293, 'acc_train': 0.93999999999999995, 'acc_test': 0.8269762299613
046, 'f_train': 0.9375, 'f_test': 0.64869193806727177}, 1: {'train_tim
e': 0.7014918327331543, 'pred_time': 0.020573854446411133, 'acc_train':
0.8833333333333333, 'acc_test': 0.85583195135433943, 'f_train': 0.81349
206349206349, 'f_test': 0.72137110106910607}, 2: {'train_time': 7.77152
9912948608, 'pred_time': 0.01898503303527832, 'acc_train': 0.8566666666
6666669, 'acc_test': 0.86301824212271971, 'f_train': 0.7341269841269840
7, 'f_test': 0.73953385618027191}}}
```

```
/Users/iqt027/projects/training/machine-learning/finding_donors/visual
s.py:75: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  ax[j/3, j%3].bar(i+k*bar_width, results[learner][i][metric], width =
bar_width, color = colors[k])
/Users/iqt027/projects/training/machine-learning/finding_donors/visual
s.py:76: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  ax[j/3, j%3].set_xticks([0.45, 1.45, 2.45])
/Users/iqt027/projects/training/machine-learning/finding_donors/visual
s.py:77: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  ax[j/3, j%3].set_xticklabels(["1%", "10%", "100%"])
/Users/iqt027/projects/training/machine-learning/finding_donors/visual
s.py:78: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  ax[j/3, j%3].set_xlabel("Training Set Size")
/Users/iqt027/projects/training/machine-learning/finding_donors/visual
s.py:79: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  ax[j/3, j%3].set_xlim((-0.1, 3.0))
```



Improving Results

In this final section, you will choose from the three supervised learning models the *best* model to use on the student data. You will then perform a grid search optimization for the model over the entire training set (`X_train` and `y_train`) by tuning at least one parameter to improve upon the untuned model's F-score.

Question 3 - Choosing the Best Model

Based on the evaluation you performed earlier, in one to two paragraphs, explain to CharityML *which of the three models you believe to be most appropriate for the task of identifying individuals that make more than $50,000.*
Hint: Your answer should include discussion of the metrics, prediction/training time, and the algorithm's suitability for the data.

Answer:

Among the three models Decision Tree, Random Forest, and Gradient Boosting, I believe that the Gradient Boosting is most appropriate for the task for the following reasons:

- According to the testing accuracy and F-score, Gradient Boosting is slightly better than Random Forest in prediction performance.
- In addition, Gradient Boosting is much faster than Random Forest in prediction even though it is slower in model training. I value prediction time performance over model training time performance because model training is relatively infrequent, while prediction time perform will directly impact daily productivity after model deployment.

Question 4 - Describing the Model in Layman's Terms

In one to two paragraphs, explain to CharityML*, in layman's terms, how the final model chosen is supposed to work. Be sure that you are describing the major qualities of the model, such as how the model is trained and how the model makes a prediction. Avoid using advanced mathematical or technical jargon, such as describing equations or discussing the algorithm implementation.*

Answer:

Gradient boosting combines several simple prediction models to create an "ensemble" of models that can make more accurate predictions.

In our case, the model is trained with existing census data (e.g., age, gender, etc), and uses that data to find rules that separate those with income above or below 50K.

To train a model, we split a given dataset into two parts (datasets), one for model training and the other for testing. The training dataset is used to train the model and then the testing dataset is used to evaluare the prediction performance of the trained model.

During successive rounds of training, new rules (i.e., decision trees) are learned that essentially create a flowchart of yes/no questions that eventually leads to a final yes/no decision at a leaf node of the trees.

With each round, the model looks at where a learned tree predicted poorly and then add a new tree to improve decisions.

The trees are then combined to create the final model that looks at new unknown individuals by feeding their information through the final model for prediction.

Implementation: Model Tuning

Fine tune the chosen model. Use grid search (`GridSearchCV`) with at least one important parameter tuned with at least 3 different values. You will need to use the entire training set for this. In the code cell below, you will need to implement the following:

- Import `sklearn.grid_search.GridSearchCV` (http://scikit-learn.org/0.17/modules/generated/sklearn.grid_search.GridSearchCV.html) and `sklearn.metrics.make_scorer` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- Initialize the classifier you've chosen and store it in `clf`.
  - Set a `random_state` if one is available to the same state you set before.
- Create a dictionary of parameters you wish to tune for the chosen model.
  - Example: `parameters = {'parameter' : [list of values]}`.
  - **Note:** Avoid tuning the `max_features` parameter of your learner if that parameter is available!
- Use `make_scorer` to create an `fbeta_score` scoring object (with $\beta = 0.5$).
- Perform grid search on the classifier `clf` using the `'scorer'`, and store it in `grid_obj`.
- Fit the grid search object to the training data (`X_train`, `y_train`), and store it in `grid_fit`.

Note: Depending on the algorithm chosen and the parameter list, the following implementation may take some time to run!

In [29]:
```
# TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary li
braries
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import make_scorer

# TODO: Initialize the classifier
clf = GradientBoostingClassifier(random_state=0)

# TODO: Create the parameters list you wish to tune
parameters = {'max_depth':[2, 5, 10]}

# TODO: Make an fbeta_score scoring object
scorer = make_scorer(fbeta_score, beta=0.5)

# TODO: Perform grid search on the classifier using 'scorer' as the scor
ing method
grid_obj = GridSearchCV(estimator=clf, param_grid=parameters, scoring=sc
orer)

# TODO: Fit the grid search object to the training data and find the opt
imal parameters
grid_fit = grid_obj.fit(X_train, y_train)

print("Best params\n------")
print(grid_fit.best_params_)

# Get the estimator
best_clf = grid_fit.best_estimator_

# Make predictions using the unoptimized and model
predictions = (clf.fit(X_train, y_train)).predict(X_test)
best_predictions = best_clf.predict(X_test)

# Report the before-and-afterscores
print ("\nUnoptimized model\n------")
print ("Accuracy score on testing data: {:.4f}".format(accuracy_score(y_
test, predictions)))
print ("F-score on testing data: {:.4f}".format(fbeta_score(y_test, pred
ictions, beta = 0.5)))
print ("\nOptimized Model\n------")
print ("Final accuracy score on the testing data: {:.4f}".format(accurac
y_score(y_test, best_predictions)))
print ("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_
test, best_predictions, beta = 0.5)))
```

```
Best params
------
{'max_depth': 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: 0.8685
Final F-score on the testing data: 0.7477
```

Question 5 - Final Model Evaluation

*What is your optimized model's accuracy and F-score on the testing data? Are these scores better or worse than the unoptimized model? How do the results from your optimized model compare to the naive predictor benchmarks you found earlier in **Question 1**?*
Note: Fill in the table below with your results, and then provide discussion in the **Answer** box.

Results:

Metric	Benchmark Predictor	Unoptimized Model	Optimized Model
Accuracy Score	0.2478	0.8630	0.8685
F-score	0.2917	0.7395	0.7477

Answer:

The optimized model's accuracy and F-score on the testing data are 0.8685 and 0.7477 respectively. These scores are slightly better than the unoptimized model. The results from the opitimized model are significantly better than the corresponding results from the naive predictor benchmarks.

Feature Importance

An important task when performing supervised learning on a dataset like the census data we study here is determining which features provide the most predictive power. By focusing on the relationship between only a few crucial features and the target label we simplify our understanding of the phenomenon, which is most always a useful thing to do. In the case of this project, that means we wish to identify a small number of features that most strongly predict whether an individual makes at most or more than $50,000.

Choose a scikit-learn classifier (e.g., adaboost, random forests) that has a `feature_importance_` attribute, which is a function that ranks the importance of features according to the chosen classifier. In the next python cell fit this classifier to training set and use this attribute to determine the top 5 most important features for the census dataset.

Question 6 - Feature Relevance Observation

When **Exploring the Data**, it was shown there are thirteen available features for each individual on record in the census data.
Of these thirteen records, which five features do you believe to be most important for prediction, and in what order would you rank them and why?

Answer:

I believe that the following five features are most important in the specified order for prediction:

- education_level
- education-num
- occupation
- workclass
- marital-status

I put education level at the top because it directly impacts the starting salary. The education-num takes the second place since it is closely related to education level. After that is occupation since payment is different for different occupation. Workclass is related to occupation and thus I put it adjacent to occupation. Finally is marital-status since payment should not heavily depend on it.

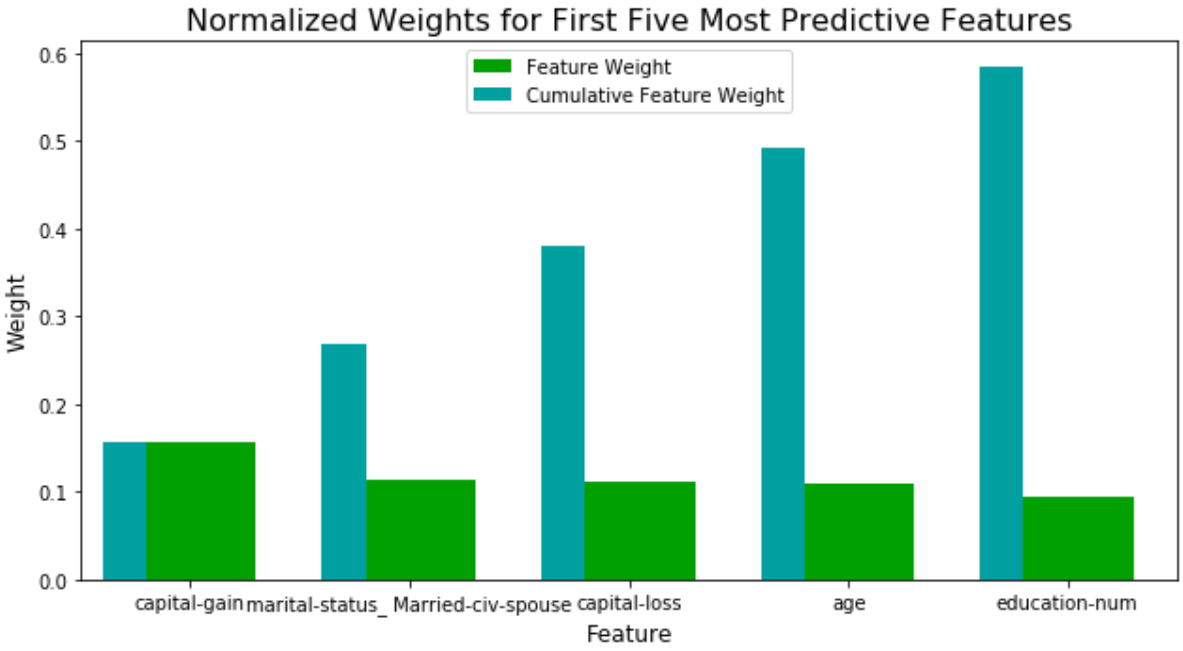
Implementation - Extracting Feature Importance

Choose a `scikit-learn` supervised learning algorithm that has a `feature_importance_` attribute availble for it. This attribute is a function that ranks the importance of each feature when making predictions based on the chosen algorithm.

In the code cell below, you will need to implement the following:

- Import a supervised learning model from sklearn if it is different from the three used earlier.
- Train the supervised model on the entire training set.
- Extract the feature importances using `'.feature_importances_'`.

In [30]:
```
# TODO: Import a supervised learning model that has 'feature_importances
_'
from sklearn.ensemble import GradientBoostingClassifier

# TODO: Train the supervised model on the training set
#
# per review comments, just use the best_clf model to extract the featur
e importances
#
# model = GradientBoostingClassifier(random_state=0)
# model.fit(X_train, y_train)

# TODO: Extract the feature importances
importances = best_clf.feature_importances_

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



Question 7 - Extracting Feature Importance

Observe the visualization created above which displays the five most relevant features for predicting if an individual makes at most or above $50,000.
*How do these five features compare to the five features you discussed in **Question 6**? If you were close to the same answer, how does this visualization confirm your thoughts? If you were not close, why do you think these features are more relevant?*

Answer:

According to the visualization, the five most relevant features are:

- capital-gain
- marital-status: civ-spouse
- capital-loss
- age
- education-num

My selected features are not close to the auto-selected five features above. There auto-selected features are more relevant because

- they have lower granularity marital-status and thus more selective.
- the auto-selected features such as capital-gain and capital-loss are more closely related to total income.
- the auto-selected feature age impacts income more than I expect.

Feature Selection

How does a model perform if we only use a subset of all the available features in the data? With less features required to train, the expectation is that training and prediction time is much lower — at the cost of performance metrics. From the visualization above, we see that the top five most important features contribute more than half of the importance of **all** features present in the data. This hints that we can attempt to *reduce the feature space* and simplify the information required for the model to learn. The code cell below will use the same optimized model you found earlier, and train it on the same training set *with only the top five important features*.

In [31]:
```
# Import functionality for cloning a model
from sklearn.base import clone

# Reduce the feature space
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]]]

# Train on the "best" model found from grid search earlier
clf = (clone(best_clf)).fit(X_train_reduced, y_train)

# Make new predictions
reduced_predictions = clf.predict(X_test_reduced)

# Report scores from the final model using both versions of data
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, beta = 0.5)))
print ("\nFinal Model trained on reduced data\n------")
print ("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test,
reduced_predictions)))
print ("F-score on testing data: {:.4f}".format(fbeta_score(y_test, redu
ced_predictions, beta = 0.5)))
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8685
F-score on testing data: 0.7477

Final Model trained on reduced data
------
Accuracy on testing data: 0.8583
F-score on testing data: 0.7240
```

Question 8 - Effects of Feature Selection

How does the final model's F-score and accuracy score on the reduced data using only five features compare to those same scores when all features are used?
If training time was a factor, would you consider using the reduced data as your training set?

Answer:

The final model's F-score and accuracy score on the reduced data using only five features are close to those same scores when all features are used. If training time was a factor, I would consider using the reduced data as the training set.

Note: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.