Data Scientist Nanodegree

Supervised Learning

Project: Finding Donors for *CharityML*

Welcome to the first project of the Data Scientist 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 [41]:

```
# 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 DataFra

# 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	sex	cap
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	21

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 [42]:

```
data.income.value_counts()
```

Out[42]:

```
<=50K    34014
>50K     11208
Name: income, dtype: int64
```

In [43]:

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

# TODO: Number of records where individual's income is more than $50,000
n_greater_50k = data[data.income=='>50K'].shape[0]

# TODO: Number of records where individual's income is at most $50,000
n_at_most_50k = data[data.income=='<=50K'].shape[0]

# TODO: Percentage of individuals whose income is more than $50,000
greater_percent = n_greater_50k / (n_greater_50k + n_at_most_50k) * 100

# 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: {}%".format(greate
```

```
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: 24.78439697492371%
```

Featureset Exploration

- **age**: continuous.
- **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**: continuous.
- **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**: continuous.
- **capital-loss**: continuous.
- **hours-per-week**: continuous.

- **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.

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.

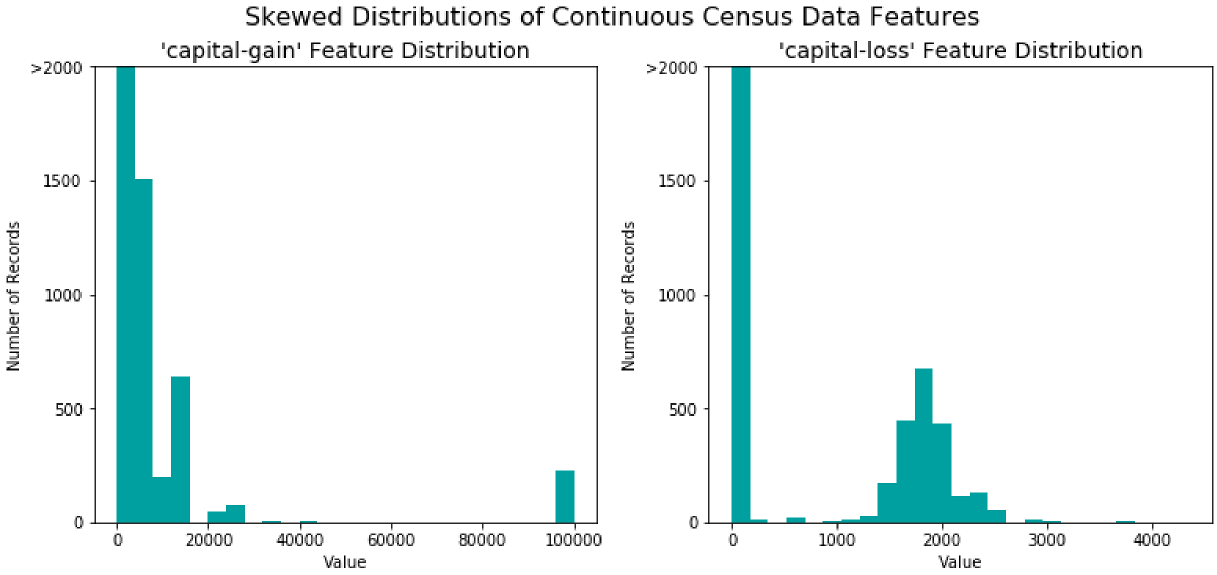
Transforming Skewed Continuous Features

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 [44]:

```python
# 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)
```

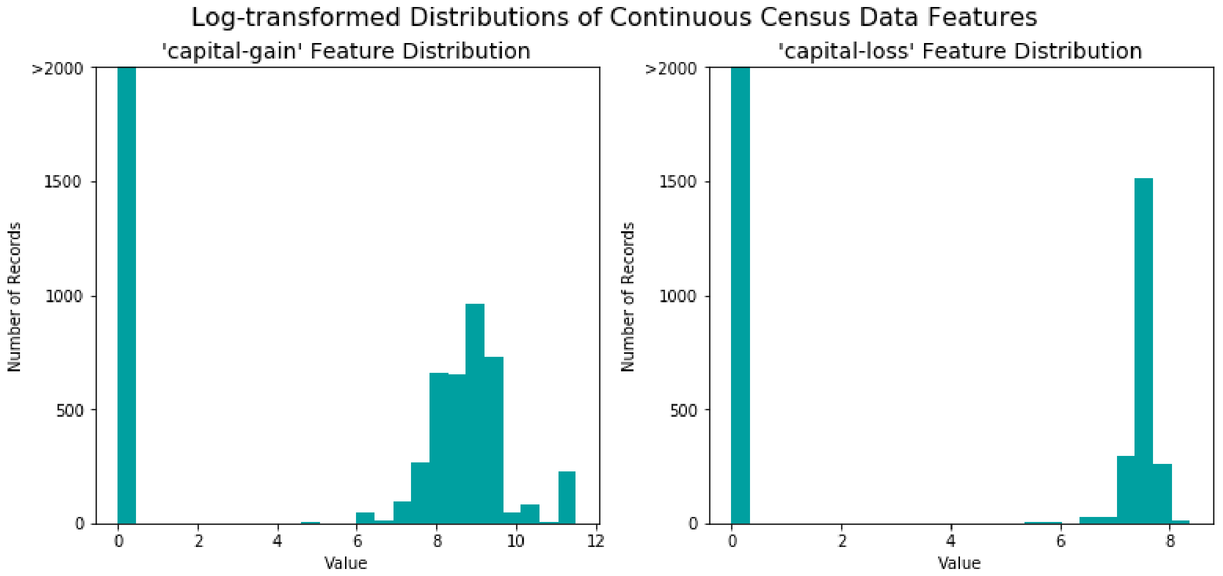


For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'` , it is common practice to apply a logarithmic transformation (https://en.wikipedia.org/wiki/Data_transformation_(statistics)) on the data so that the very large and very small values do not negatively affect the performance 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 [45]:

```python
# Log-transform the skewed features
skewed = ['capital-gain', 'capital-loss']
features_log_transformed = pd.DataFrame(data = features_raw)
features_log_transformed[skewed] = features_raw[skewed].apply(lambda x: np.lo

# Visualize the new log distributions
vs.distribution(features_log_transformed, 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)](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html) for this.

In [46]:

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

# Initialize a scaler, then apply it to the features
scaler = MinMaxScaler() # default=(0, 1)
numerical = ['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-p

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

# Show an example of a record with scaling applied
display(features_log_minmax_transform.head(n = 5))
```

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

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_log_minmax_transform'` data.
- Convert the target label `'income_raw'` to numerical entries.
  - Set records with "<=50K" to `0` and records with ">50K" to `1`.

In [47]:

```
income_raw.value_counts()
```

Out[47]:

```
<=50K    34014
>50K     11208
Name: income, dtype: int64
```

In [48]:

```
# TODO: One-hot encode the 'features_log_minmax_transform' data using pandas.
features_final = pd.get_dummies(features_log_minmax_transform)

# TODO: Encode the 'income_raw' data to numerical values
income = income_raw.map({'<=50K':0, '>50K':1})

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

# Uncomment the following line to see the encoded feature names
#print(encoded)
display(features_final.head(5))
```

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

	age	education-num	capital-gain	capital-loss	hours-per-week	workclass_ Federal-gov	workclass_ Local-gov	workclass_ Private	workclas Self-er
0	0.301370	0.800000	0.667492	0.0	0.397959	0	0	0	
1	0.452055	0.800000	0.000000	0.0	0.122449	0	0	0	
2	0.287671	0.533333	0.000000	0.0	0.397959	0	0	1	
3	0.493151	0.400000	0.000000	0.0	0.397959	0	0	1	
4	0.150685	0.800000	0.000000	0.0	0.397959	0	0	1	

5 rows × 103 columns

In [49]:

```
income.value_counts()
```

Out[49]:

```
0    34014
1    11208
Name: income, dtype: int64
```

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 [50]:

```python
# Import train_test_split
from sklearn.cross_validation 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_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.
```

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.

Note: Recap of accuracy, precision, recall

Accuracy measures how often the classifier makes the correct prediction. It’s the ratio of the number of correct predictions to the total number of predictions (the number of test data points).

Precision tells us what proportion of messages we classified as spam, actually were spam. It is a ratio of true positives(words classified as spam, and which are actually spam) to all positives(all words classified as spam, irrespective of whether that was the correct classificatio), in other words it is the ratio of

```
[True Positives/(True Positives + False Positives)]
```

Recall(sensitivity) tells us what proportion of messages that actually were spam were classified by us as spam. It is a ratio of true positives(words classified as spam, and which are actually spam) to all the words that were actually spam, in other words it is the ratio of

```
[True Positives/(True Positives + False Negatives)]
```

For classification problems that are skewed in their classification distributions like in our case, for example if we had a 100 text messages and only 2 were spam and the rest 98 weren't, accuracy by itself is not a very good metric. We could classify 90 messages as not spam(including the 2 that were spam but we classify them as not spam, hence they would be false negatives) and 10 as spam(all 10 false positives) and still get a reasonably good accuracy score. For such cases, precision and recall come in very handy. These two metrics can be combined to get the F1 score, which is weighted average(harmonic mean) of the precision and recall scores. This score can range from 0 to 1, with 1 being the best possible F1 score(we take the harmonic mean as we are dealing with ratios).

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? You must use the code cell below and assign your results to `'accuracy'` and `'fscore'` to be used later.

Please note that the the purpose of generating a naive predictor is simply to show what a base model without any intelligence would look like. In the real world, ideally your base model would be either the results of a previous model or could be based on a research paper upon which you are looking to improve. When there is no benchmark model set, getting a result better than random choice is a place you could start from.

HINT:

- When we have a model that always predicts '1' (i.e. the individual makes more than 50k) then our model will have no True Negatives(TN) or False Negatives(FN) as we are not making any negative('0' value) predictions. Therefore our Accuracy in this case becomes the same as our Precision(True Positives/(True Positives + False Positives)) as every prediction that we have made with value '1' that should have '0' becomes a False Positive; therefore our denominator in this case is the total number of records we have in total.
- Our Recall score(True Positives/(True Positives + False Negatives)) in this setting becomes 1 as we have no False Negatives.

In [51]:

```

TP = np.sum(income) # Counting the ones as this is the naive case. Note that
# encoded to numerical values done in the data preprocessing step.
FP = income.count() - TP # Specific to the naive case

TN = 0 # No predicted negatives in the naive case
FN = 0 # No predicted negatives in the naive case

# TODO: Calculate accuracy, precision and recall
accuracy = TP/income.shape[0]
recall = TP/(TP+FN)
precision = TP/(TP+FP)

# TODO: Calculate F-score using the formula above for beta = 0.5 and correct
beta = 0.5
fscore = (1+beta**2)*(precision*recall)/((beta**2 * precision)+recall)

# Print the results
print("Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".format(acc
```

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

Supervised Learning Models

The following are some of the supervised learning models that are currently available in `scikit-learn` [(http://scikit-learn.org/stable/supervised_learning.html)](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.
- 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?

HINT:

Structure your answer in the same format as above^, with 4 parts for each of the three models you pick. Please include references with your answer.

Answer:

1) Logistic Regression:

- Could be used effectively to analyze a bag of words problem, such as classifiying an email as spam.
- Simple probabalistic interpretation, and the algorithm can be regularized to avoid overfitting. Logistic regression models can be fit relatively inexpensively.
- May not be flexible enough to interpret multiple nonlinear decision boundaries.
- This is a good first choice because it is inexpensive to implement, and because we have already normalized our data. It will avoid overfitting the data and scale well with a large dataset. Since we do not know the impact of each feature yet, it will provide a good contrast to decision-tree based methods later.
- references: Elite Data Science (https://elitedatascience.com/machine-learning-algorithms#classification), Quora (https://www.quora.com/What-are-applications-of-linear-and-logistic-regression)

2) AdaBoost:

- Could be applied in a scenario where one is trying to predict if a user profile is prone to defaulting on a loan.
- Robust to outliers, and model multiple nonlinear decision boundaries well.
- Prone to overfitting if not constrained well, as it is built on decision tree weak learners.
- I've chosen this as a second model to try because it excels in modeling nonlinear feature interactions, which is a weakness of the logistic regression tested prior. Since our data has many features, it should be an effective model.
- references: Elite Data Science (https://elitedatascience.com/machine-learning-algorithms#classification), Wikipedia (https://en.wikipedia.org/wiki/AdaBoost)

3) Random Forest:

- Could be applied to identify a patient's illness by analyzing their medical record.
- As an ensemble method, they naturally perform well on datasets with a high number of features, and the Random Forest algorithm avoids overfitting better than other ensemble methods. Random forests also are less memory intensive than some other models.
- Sensitive to overfitting if a dataset is particularly noisy.
- I've chosen this method because we are working with a large dataset comprised of many features, and I want to continue using non-memory-intensive models. As an ensemble method it should perform well with our featureset, and may be less prone to overfitting than AdaBoost.
- references: newgenapps (https://www.newgenapps.com/blog/random-forest-analysis-in-ml-and-when-to-use-it), CitizenNet (http://blog.citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics), Wikipedia (https://en.wikipedia.org/wiki/Random_forest)

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)](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 [52]:

```
X_test.sample(n=1)
```

Out[52]:

	age	education-num	capital-gain	capital-loss	hours-per-week	workclass_ Federal-gov	workclass_ Local-gov	workclass_ Private	wor S
14468	0.506849	0.533333	0.707941	0.0	0.397959	0	0	1	

1 rows × 103 columns

In [53]:

```python
# 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
       - 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_s
    #       using .fit(training_features[:], training_labels[:])
    start = time() # Get start time
    # learner.fit(X_train.sample(n=sample_size), y_train.sample(n=sample_size
    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) u
    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_tra
    results['acc_train'] = accuracy_score(y_train[:300], predictions_train[:3

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

    # TODO: Compute F-score on the the first 300 training samples using fbeta
    results['f_train'] = fbeta_score(y_train[:300], predictions_train[:300],

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

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

    # Return the results
    return results
```

Implementation: Initial Model Evaluation

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

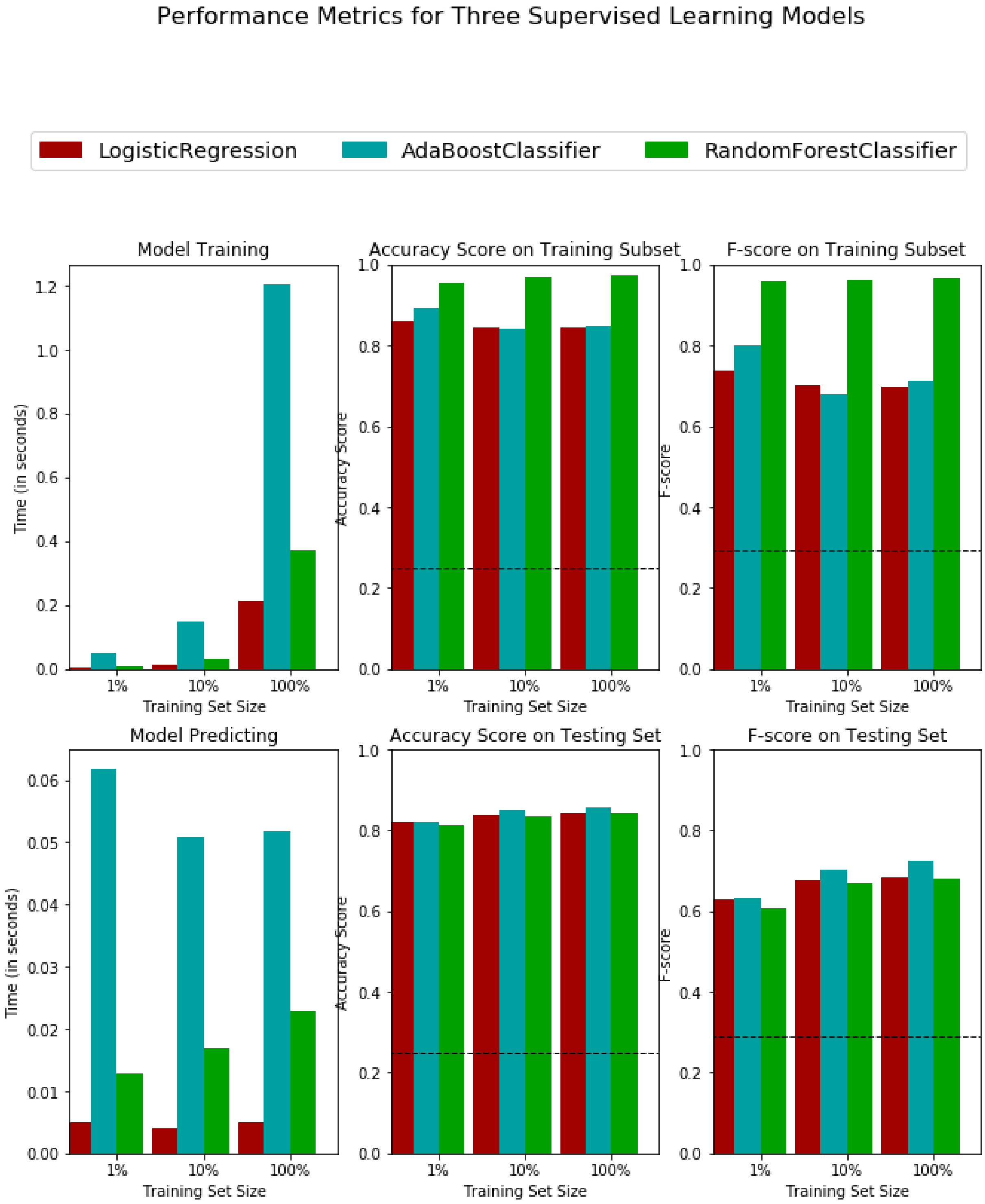
- Import the three supervised learning models you've discussed in the previous section.
- Initialize the three models and store them in `'clf_A'` , `'clf_B'` , and `'clf_C'` .
  - Use a `'random_state'` for each model you use, if provided.
  - **Note:** Use the default settings for each model — you will 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 [54]:

```
# TODO: Import the three supervised learning models from sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier

# TODO: Initialize the three models
clf_A = LogisticRegression()
clf_B = AdaBoostClassifier(random_state=13)
clf_C = RandomForestClassifier()

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the training
# HINT: samples_100 is the entire training set i.e. len(y_train)
# HINT: samples_10 is 10% of samples_100 (ensure to set the count of the valu
# HINT: samples_1 is 1% of samples_100 (ensure to set the count of the values
samples_100 = len(y_train)
samples_10 = int(len(y_train)/10)
samples_1 = int(len(y_train)/100)

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

```
LogisticRegression trained on 361 samples.
LogisticRegression trained on 3617 samples.
LogisticRegression trained on 36177 samples.
AdaBoostClassifier trained on 361 samples.
AdaBoostClassifier trained on 3617 samples.
AdaBoostClassifier trained on 36177 samples.
RandomForestClassifier trained on 361 samples.
RandomForestClassifier trained on 3617 samples.
RandomForestClassifier trained on 36177 samples.
```



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: Look at the graph at the bottom left from the cell above(the visualization created by `vs.evaluate(results, accuracy, fscore)`) and check the F score for the testing set when 100% of the training set is used. Which model has the highest score? Your answer should include discussion of the:

- metrics - F score on the testing when 100% of the training data is used,
- prediction/training time
- the algorithm's suitability for the data.

Answer:
I believe that the AdaBoost model is going to be most appropriate for the task, as it has the highest F-score on the testing data when 100% of the data is used. Although the AdaBoost is the most time intensive by far, it is still remarkably fast, and the training time is negligible. Note that the AdaBoost classifier is outperformed by the Random Forest classifier in terms of F-score and accuracy on the training set, but then it outperforms the Random Forest classifier on the testing sets. The default random forest classifier is overfitting the training data, whereas the AdaBoost classifier's f-scores on the testing and training datasets are comparable, suggesting it is not overfitting. Decision trees like AdaBoost and Random Forest are well suited to problems with many features such as this one, and it makes sense to choose the one that does not overfit by default.

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 jargon, such as describing equations.

HINT:

When explaining your model, if using external resources please include all citations.

Answer:
The AdaBoost classifier is what is known as an ensemble method, meaning that is actually a classifier comprised of many 'weak learners', or simple classifiers, which it trains on subsets of the data and then has vote on classification when asked to predict an outcome. What makes the AdaBoost algorithm unique is that it iteratively trains its weak learners on the dataset, adjusting the impact of misclassification each training round to push each new weak learner to improve on the mistakes of all the learners before it. Finally, it assigns each weak learner a weight based on the accuracy it achieved which impacts how valuable its vote is when a prediction is made. As a decision tree, AdaBoost is well suited for handling many complex features as we have in this program, and as an ensemble method it uses the concept of voting to try to reduce the potential for memorizing the training data.

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 [55]:

```
# TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary librari
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

# TODO: Initialize the classifier
clf = AdaBoostClassifier(random_state=13)

# TODO: Create the parameters list you wish to tune, using a dictionary if ne
# HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1
parameters = {'n_estimators':[50, 100, 300, 500], \
              'learning_rate':[1., 1.25, 1.5, 1.75, 2.]}

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

# TODO: Perform grid search on the classifier using 'scorer' as the scoring m
grid_obj = GridSearchCV(clf, parameters, scoring=scorer)

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

# 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("Unoptimized model\n------")
print("Accuracy score on testing data: {:.4f}".format(accuracy_score(y_test,
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, prediction
print("\nOptimized Model\n------")
print("Final accuracy score on the testing data: {:.4f}".format(accuracy_scor
print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test,
```

```
C:\Users\Jesse\Anaconda3\lib\site-packages\sklearn\metrics\classification.py:11
35: UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to
no predicted samples.
  'precision', 'predicted', average, warn_for)
C:\Users\Jesse\Anaconda3\lib\site-packages\sklearn\metrics\classification.py:11
35: UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to
no predicted samples.
  'precision', 'predicted', average, warn_for)
C:\Users\Jesse\Anaconda3\lib\site-packages\sklearn\metrics\classification.py:11
35: UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to
no predicted samples.
  'precision', 'predicted', average, warn_for)
C:\Users\Jesse\Anaconda3\lib\site-packages\sklearn\metrics\classification.py:11
35: UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to
no predicted samples.
  'precision', 'predicted', average, warn_for)

Unoptimized model
------
Accuracy score on testing data: 0.8576
F-score on testing data: 0.7246
```

```
Optimized Model
------
Final accuracy score on the testing data: 0.8677
Final F-score on the testing data: 0.7452
```

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	Unoptimized Model	Optimized Model
Accuracy Score	0.8576	0.8677
F-score	0.7246	0.7452

```
In [56]: grid_fit.best_estimator_
```

```
Out[56]: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
                   learning_rate=1.5, n_estimators=500, random_state=13)
```

Answer:
Both the accuracy score and F-score improved in the optimized model, as expected. The improvements were marginal, and the best estimator's parameters indicated that increasing the number of weak learners led to an increase inscoring metrics (with diminishing returns), although increasing the learning rate did not necessarily.

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?

In [57]:
```
data['education-num'].value_counts()
```

Out[57]:
```
9.0     14783
10.0     9899
13.0     7570
14.0     2514
11.0     1959
7.0      1619
12.0     1507
6.0      1223
4.0       823
15.0      785
5.0       676
8.0       577
16.0      544
3.0       449
2.0       222
1.0        72
Name: education-num, dtype: int64
```

In [58]:
```
data.columns
```

Out[58]:
```
Index(['age', 'workclass', 'education_level', 'education-num',
       'marital-status', 'occupation', 'relationship', 'race', 'sex',
       'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
       'income'],
      dtype='object')
```

In [59]:
```
data['race'].value_counts()
```

Out[59]:
```
 White                 38903
 Black                  4228
 Asian-Pac-Islander     1303
 Amer-Indian-Eskimo      435
 Other                   353
Name: race, dtype: int64
```

Answer:
There are several features in this dataset that I expect to contribute heavily towards an individual's annual earnings.

1) **Captial Losses:** This will obviously directly impact an individuals net earnings, as if they are paying off loans or other debt their yearly takeaway will diminish heavily. I expect many surveyed also own depreciating assets, such as cars.

2) **Age:** Numerous studies have shown that an individual's earning potential increases with age up

to a threshold value. Persons between 35-40 have a higher earning potential than persons 25-30 because they have more industry experience and job skills.

3) **Captial Gains:** This will also directly impact an individuals net earnings, but I suspect fewer people are profiting from the sales of assets than are losing from depreciating assets (captial losses) so it will play a smaller role. Those who are profiting from captial gains are also more likely to be older, which puts this feature behind age in terms of importance.

4) **Hours per Week:** Very simply, any individual working a full 40 hours per week is more likely to have a higher income than one working part time at 20-30 hours per week. A part time worker is much more likely to be working at an hourly rate as opposed to a salary as well, which in all likelihood decreases their earning ceiling. I put this 4th on the list because I suspect more of those surveyed are probably working 40 hours per week, making this more of an edge case.

5) **Education-num:** Numerous studies have also shown that a person's earning potential is correlated with their education level. The only reason I place this last on the list is because I think that once regularized, the difference between education levels may be misleadingly small.

Implementation - Extracting Feature Importance

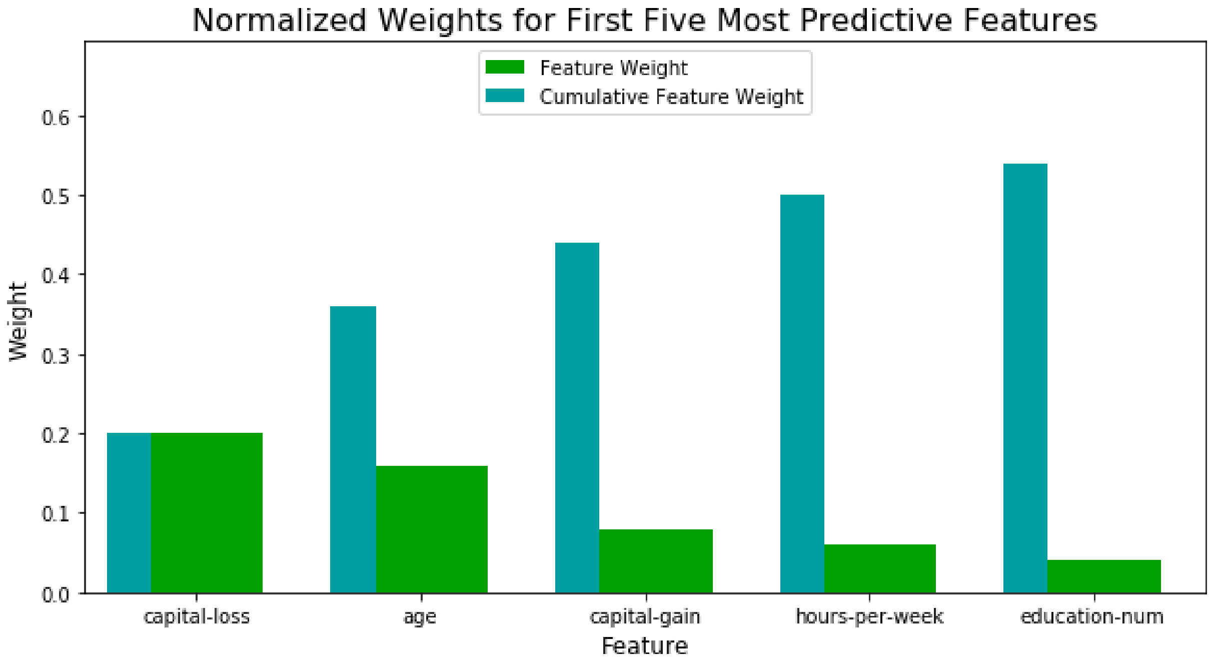
Choose a `scikit-learn` supervised learning algorithm that has a `feature_importance_` attribute availble for it. This attribute is a function that ranks the importance of each feature when making predictions based on the chosen algorithm.

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 [60]:

```
# TODO: Import a supervised learning model that has 'feature_importances_'


# TODO: Train the supervised model on the training set using .fit(X_train, y_
model = AdaBoostClassifier().fit(X_train, y_train)

# TODO: Extract the feature importances using .feature_importances_
importances = model.feature_importances_

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

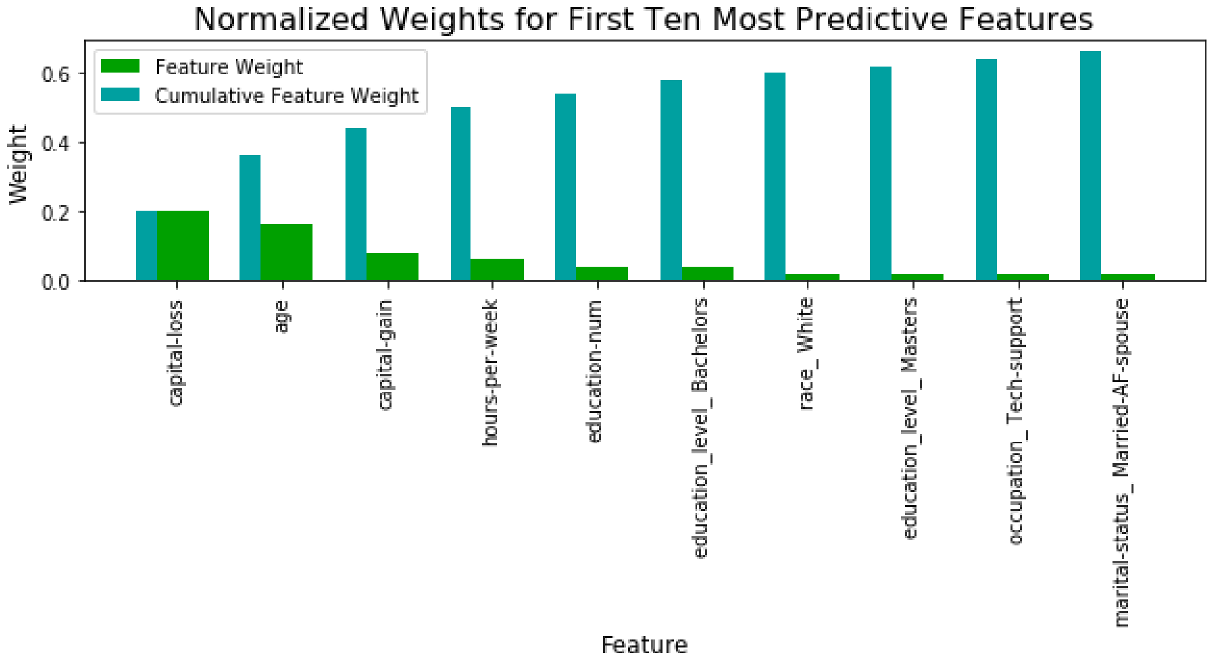


In [61]:

```
import matplotlib.pyplot as plt
def feature_plot2(importances, X_train, y_train):

    # Display the five most important features
    indices = np.argsort(importances)[::-1]
    columns = X_train.columns.values[indices[:10]]
    values = importances[indices][:10]

    # Creat the plot
    fig = plt.figure(figsize = (9,5))
    plt.title("Normalized Weights for First Ten Most Predictive Features", fo
    plt.bar(np.arange(10), values, width = 0.6, align="center", color = '#00A
          label = "Feature Weight")
    plt.bar(np.arange(10) - 0.3, np.cumsum(values), width = 0.2, align = "cen
          label = "Cumulative Feature Weight")
    plt.xticks(np.arange(10), columns, rotation=90)
    #plt.xlim((-0.5, 4.5))
    plt.ylabel("Weight", fontsize = 12)
    plt.xlabel("Feature", fontsize = 12)

    plt.legend(loc = 'upper left')
    plt.tight_layout()
    plt.show()
```

In [62]:

```
feature_plot2(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:
This answer confirms my suspicions exactly. The top 5 features I predicted together account for over 50% of the model weight. If we expand the list, we can see that race_White makes it into the top 7 which is not too surprising, I suspect it is only that low on the list because the vast majority of the sampled population is white. The same logit applies for why the education_level_Masters is lower on the list than education_level_Bacherlors - although Master's degrees increase earning potential drastically, many more participants have a bachelor's degree or lower than have a master's.

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 [63]:

```python
# 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)[::-
X_test_reduced = X_test[X_test.columns.values[(np.argsort(importances)[::-1])

# 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_p
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, best_predi
print("\nFinal Model trained on reduced data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduce
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_pr
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8677
F-score on testing data: 0.7452

Final Model trained on reduced data
------
Accuracy on testing data: 0.8421
F-score on testing data: 0.7003
```

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?

In [68]:
```
cy reduction: {}'.format(1 - accuracy_score(y_test, reduced_predictions)/accuracy_
e reduction: {}'.format(1 - fbeta_score(y_test, reduced_predictions, beta = 0.5)/f
```

```
Accuracy reduction: 0.029434250764525993
F-score reduction: 0.060259697099178666
```

Answer:
Accuracy was reduced by 3%, and the F-score was reduce by 6%. Deciding to reduce the training set data poses a cost-benefit question that would have to be analyzed on its own before pursuing either option. It is possible that opting to use a full dataset at the cost of time could pose direct costs via overhead, and indirect costs via schedule delays or missed opportunities. However, it could be that a small reduction in precision or recall could translate into a reduction in ROI which would scale over time, or it could mean missed fraud or medical diagnoses immediately which would pose issues for a business.

In this case, the reduction in f-score would translate into misclassifying a few hundred potential donors as earning >50K/year, and would result in a marginally lower return on investment in promotional material and postage. Running my optimized model on the full dataset only takes a few minutes, so it is worth it to run the complete model. However, if the client decided to use a larger dataset which could delay my model fitting by days or weeks, it would likely be worth the tradeoff.

> **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.

In [ ]:
```

```