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:** 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	Whit

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 [2]:
```
# TODO: Total number of records
n_records = data.shape[0]

# TODO: Number of records where individual's income is more than $50,0
00
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*100.0/n_records

# 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}%".fo
rmat(greater_percent)
```

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

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.

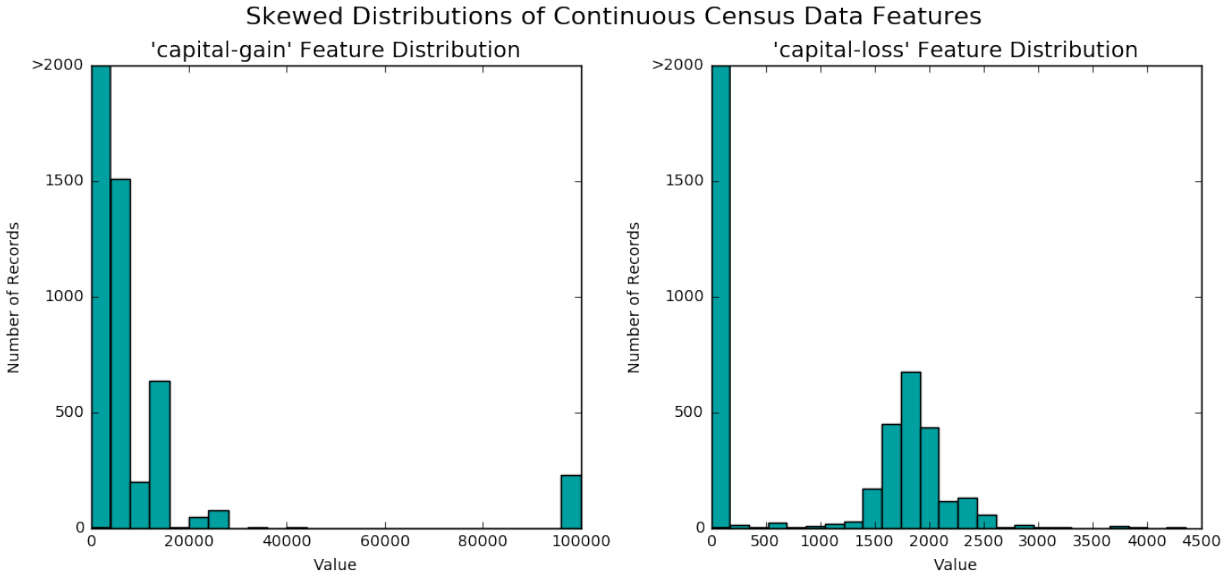
Transforming Skewed Continuous Features

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 [3]:
```
# 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)
```

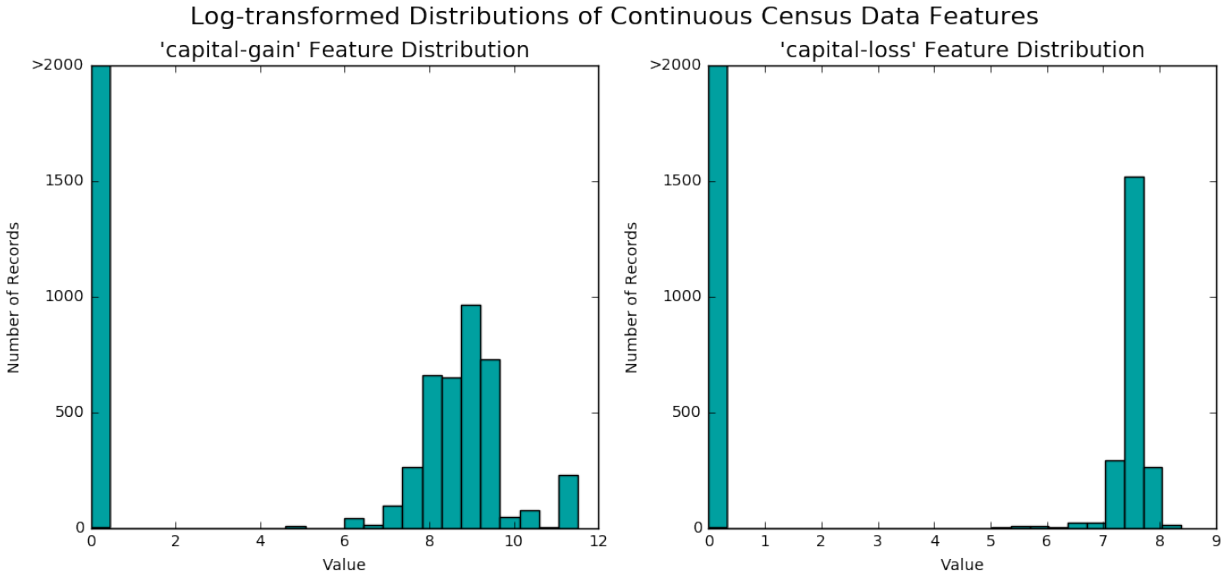


For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'`, it is common practice to apply a logarithmic transformation (https://en.wikipedia.org/wiki/Data_transformation_(statistics)) on the data so that the very large and very small values do not negatively affect the performance 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 [4]:
```
# 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 [5]:
```
# 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', '
hours-per-week']
features_raw[numerical] = scaler.fit_transform(data[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
0	0.30137	State-gov	Bachelors	0.8	Never-married	Adm-clerical	Not-in-family

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 [6]:
```
# TODO: One-hot encode the 'features_raw' data using pandas.get_dummies()
features = pd.get_dummies(features_raw, columns=['workclass', 'education_level', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country'])

# TODO: Encode the 'income_raw' data to numerical values
income = [1 if x == '>50K' else 0 for x 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
#print income
```

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

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 [7]:
```
# 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, 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.

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 [19]:
```
# TODO: Calculate accuracy
#from sklearn.metrics import accuracy_score
#from sklearn.metrics import precision_score
#from sklearn.metrics import recall_score
#print len(y_test)
y_pred = [1] * len(income)
#accuracy = accuracy_score(income, y_pred)
accuracy = sum([i == j for i, j in zip(income, y_pred)])*1.0/len(incom
e)
#print accuracy

# TODO: Calculate F-score using the formula above for beta = 0.5
#precision = precision_score(income, y_pred)
TP = sum([i == j for i, j in zip(income, y_pred) if i == 1])*1.0
FP = sum([i != j for i, j in zip(income, y_pred) if j == 1])*1.0
FN = sum([i != j for i, j in zip(income, y_pred) if j == 0])*1.0

precision = TP/(TP+FP)
#print precision

#recall = recall_score(income, y_pred)
recall = TP/(TP+FN)
#print recall

beta = 0.5
fscore = (1+np.square(beta))*precision*recall/((np.square(beta)*precis
ion)+recall)

# Print the results
print "Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".for
mat(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: The following are three models.

1. Decision Trees

An application where Decision Trees can be applied is for the Automobile industry. One can use the Decision Trees model to predict whether or not a car has Good/Bad Mileage Per Gallon, based on the maker, cylinders, horsepower, weight, etc. (Reference: Andrew W. Moore https://pdfs.semanticscholar.org/4049/1dc18f32292d56508729e4d66738999a1542.pdf (https://pdfs.semanticscholar.org/4049/1dc18f32292d56508729e4d66738999a1542.pdf) Data from the UCI repository (thanks to Ross Quinlan))

Strengths of the model include: (1) It can generate understanable rules (2) It can handle both categorical and continuous variables (3) It can provide a good indication on which features are important for prediction. (4) Not much computation is required. The decision trees model performs well when both categorical and continious variables are present in the data set and that much computation is not desired for performing classification.

Weaknesses of the model include: (1) It does not perform well for predicting continous variable. (2) The error rate will be large when Decision Trees are applied where there are a lot of classes and the number of training samples is relatively small. Decision Trees do not perform well for predicting continuous variables or when the number of classes is large and the number training samples is relatively small.

The Strengths and Weaknesses are described above according to Reference: Andrew W. Moore https://pdfs.semanticscholar.org/4049/1dc18f32292d56508729e4d66738999a1542.pdf (https://pdfs.semanticscholar.org/4049/1dc18f32292d56508729e4d66738999a1542.pdf)

For the problem in this project, Decision Trees can be a good candidate, as there are both categorical and continious variables for the input attributes and the rules generated from Decision Trees can be understandable.

1. Support Vector Machines (SVMs)

An application where SVMs can be applied is for the Pharmaceutical Industry. A SVMs based model can be built to predict anti-cancer drug sensitivity in cell lines based on gene expression data (Dong et al. BMC Cancer 2015 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4485860/ (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4485860/)).

Strengths of SVMs include: (1) The model can be applied on data sets with high dimensions (2) Relatively easy to train with no local optimal. SVMs can perform well when there is a lot of dimensions in the data set. (Reference: Rutgers University CS 536: Machine Learning Littman (Wu, TA) http://www.cs.rutgers.edu/~mlittman/courses/ml04/svm.pdf (http://www.cs.rutgers.edu/~mlittman/courses/ml04/svm.pdf))

Weaknesses of SVMs include: (1) SVMs can be "abysmally slow" in testing. Burges, 1996; Osuna and Girosi, 1998. Burgess (1998) (2) Further research is needed to optimally design SVMs for multi-class predicitons. Burgess (1998). SVMs may be "abysmally slow" when there are a lot of test data. (Reference for these two points: http://www.svms.org/disadvantages.html (http://www.svms.org/disadvantages.html)) (3) It is not quite transparent of the results from the SVMs as SVMs can not represent a metric based on a simple parametric function of all the data (Reference: Laura Auria, Rouslan A. Moro, Berlin, August 2008, Support Vector Machines (SVM) as a Technique for Solvency Analysis https://core.ac.uk/download/pdf/6302770.pdf (https://core.ac.uk/download/pdf/6302770.pdf)). SVMs could perform poorly when there is a large number of test data when speed is important. Also, further research is needed to optimally design SVMs for multi-class predictions.

For the problem in this project, with the number of test data is not huge and there is a decent number of input variables for precition, plus that we are making binary predictions, SVMs can be a easy-to-train model to be used for this problem.

1. Logistic Regression (LR)

An application where LR can be applied is to predict whether or not a company will default on its bonds, based on a variety of financial data (Revenue, Profits, Growth rate, Balance sheet data, etc) in the financial services industry (Reference: A)

Strengths of LR include: (1) It is a widely used and well known method in marketing (2) It is easy to implement and relatively straightforwad. (For 1 and 2, Reference A below). (3) It can output a probability value with which a spefici threshold can be applied to decide the class label suite the specific business needs (4) It is robust to small noise. (For 3 and 4, Refernce B below). LR performs well when there might be small noise in the data and a probability value is needed to be associated with the class label.

Weaknesses of LR include: (1) If the number of predicting attributes is large, the LR model might not be optimal without first reducting the predicting features by pre-processing (Reference A) (2) If there is dependence of some data points with others, LR could over-weigh the importance of those observations and resulting in bias. In these situations, LR could perform poorly.

References for LR:

A) Stephen G. Powell, Kenneth R. Baker, Management Science. Chapter 6: Classification and Prediction Methods. PowerPoint slides from: faculty.tuck.dartmouth.edu/images/uploads/faculty/management-science/Ch06.ppt

B) Lalit Sachan 2015, http://www.edvancer.in/logistic-regression-vs-decision-trees-vs-svm-part2/ (http://www.edvancer.in/logistic-regression-vs-decision-trees-vs-svm-part2/)

C) Nick Robinson http://www.ehow.com/info_8574447_disadvantages-logistic-regression.html (http://www.ehow.com/info_8574447_disadvantages-logistic-regression.html)

For this project, given that there could be small noise in the data set and the advantage of associating a probablity (then the predicted class with higher probablity could be prioritized) with the class label, Logistric Regression can be a good model to solve this problem.

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 [10]:
```
# 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 training set
       - X_train: features training set
       - y_train: income training set
       - X_test: features testing set
       - y_test: income testing set
    '''

    results = {}

    # TODO: Fit the learner to the training data using slicing with 'sample_size'
    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)
    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
    results['acc_train'] = accuracy_score(y_train[:300], predictions_t
rain)

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

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

    # TODO: Compute F-score on the test set
    results['f_test'] = fbeta_score(y_test, predictions_test, 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 [11]:
```
# TODO: Import the three supervised learning models from sklearn
from sklearn import tree
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# TODO: Initialize the three models
clf_A = tree.DecisionTreeClassifier(random_state=1)
clf_B = SVC(random_state=1)
clf_C = LogisticRegression(random_state=1)

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the t
raining data
samples_1 = X_train.shape[0]/100
#print "Training set has {} samples.".format(X_train.shape[0])
#print samples_1
samples_10 = X_train.shape[0]/10
samples_100 = X_train.shape[0]

# 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 c
hosen
vs.evaluate(results, accuracy, fscore)
```

```
DecisionTreeClassifier trained on 361 samples.
DecisionTreeClassifier trained on 3617 samples.
DecisionTreeClassifier trained on 36177 samples.

/Users/kevin/anaconda/lib/python2.7/site-packages/sklearn/metrics/cl
assification.py:1074: UndefinedMetricWarning: F-score is ill-defined
and being set to 0.0 due to no predicted samples.
  'precision', 'predicted', average, warn_for)

SVC trained on 361 samples.
SVC trained on 3617 samples.
SVC trained on 36177 samples.
LogisticRegression trained on 361 samples.
LogisticRegression trained on 3617 samples.
LogisticRegression 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: Your answer should include discussion of the metrics, prediction/training time, and the algorithm's suitability for the data.

Answer: Out of the three models, I believe the Logistic Regression model is the most appropriate for the task of identifying individuals that make more than $50,000. The resons are the following:

1. The Logistic Regression resulted in both the highest Accuracy Score and the highest F-Score on the Testing Set, among the three models, with every training set size. The Decision Tree model resulted in the highest Accuracy Score and the highest F-Score on the selected Training Set, but I believe more emphasis should be put on the accuracy score and F-Score on the test data set. Also, the accuracy score of ~ 0.84 and F-score of ~ 0.7 on the test set when the model was trained using 100\% training data are much higher than the the respective scores from the Naive Predictor with Accuracy score of 0.2478 and F-score: 0.2917.
2. The Logistic Regression model took much less time to both train and predict, while the Support Vector Machines model took extremely long time to train and predict, when all training data are used.

In summary, the Logistic Regression (LR) model is the most suitable among the three models tested. The LR model is robust to small noise in the data and with the relatively reasonable number of predicting attibutes in the data set, LR appears to be a good choice for this problem, as supported by the high prediction scores and low training and testing time in comparison with the other two chosen models.

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: The final model chosen, Logistic Regression, utilizes a statistical approach to predict whether or not a person's annual income is at most $50,000 or greater than $50,000. This is one of the machine learning based models, meaning that the model will be able to make prediction without being explicitly programmed to do so. The model accomplishes this by being trained based on a set of training data, i.e., people whose annual income, along with other attributes, such as age, workclass, education level, marital status, etc. are known. The annual income level is the variable that the model will predict for, using information from the other variables. The model assigns a weight for each predicting attribute that determines how each feature can contribute to the final classification of a person's income level. When we use the model to predict the income level of a potential donor, we combine each feature's weight with the feature's value (based on the census data) for every predicting feature. We come up with a summarized measure including all these combined values for all the features. This summarized value will then be subject to a mathematical function called the Sigmoid function, which takes as input the summarized value and output a probability value between 0 and 1. This probability value will then be used to determine whether or not a potential donor's income is greater than $50,000 (with 1 being very certain and 0 being unlikely and 0.5 being a usual threshold).

Reference: Thanks to previous reviewer's feedback. The above answer incorporated suggested descriptions from the reviewer along with the included refernce link (https://www.quora.com/What-is-logistic-regression (https://www.quora.com/What-is-logistic-regression)).

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 [12]:
```
# TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
from sklearn import grid_search
from sklearn.metrics import make_scorer

# TODO: Initialize the classifier
clf = LogisticRegression(random_state=1)

# TODO: Create the parameters list you wish to tune
parameters = {'C':np.logspace(-2,2,num=5), 'penalty':['l1', 'l2']}

# 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 scoring method
grid_obj = grid_search.GridSearchCV(clf, param_grid=parameters, scoring=scorer, cv=5)

# TODO: Fit the grid search object to the training data and find the optimal parameters
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, predictions))
print "F-score on testing data: {:.4f}".format(fbeta_score(y_test, predictions, beta = 0.5))
print "\nOptimized Model\n------"
print "Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test, best_predictions))
print "Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, beta = 0.5))
```

```
Unoptimized model
------
Accuracy score on testing data: 0.8483
F-score on testing data: 0.6993

Optimized Model
------
Final accuracy score on the testing data: 0.8494
Final F-score on the testing data: 0.7008
```

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	0.8483	0.8494
F-score	0.2917	0.6993	0.7008

Answer: The optimized model's accuracy and F-score are: 0.8494 and 0.7008, respectively. These are better than the unoptimized model. The results from my optimized model are much better than the naive predictor benchmarks. Furthermore, the improvement of the scores from the unoptimized model to the optimized model is modest. For the optimization, it may be worth including more parameters and/or more different values in the grid search to see if the model might be able to be further optimized.

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 the following are the five most important for prediction, ordered from the most important to the fifth most important: (1) occupation (2) education_level (3) workclass (4) education-num (5) age. I believe the type of occuptation makes a huge impact on the level of annual income for a person, followed by the education level, the higher educational level one has, the more likely the person could make more, followed by workclass, with certain working classes making more than the other, education number, with the number of educational years affect the income, and age, generally the senior one gets, the higher income level one will have.

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:

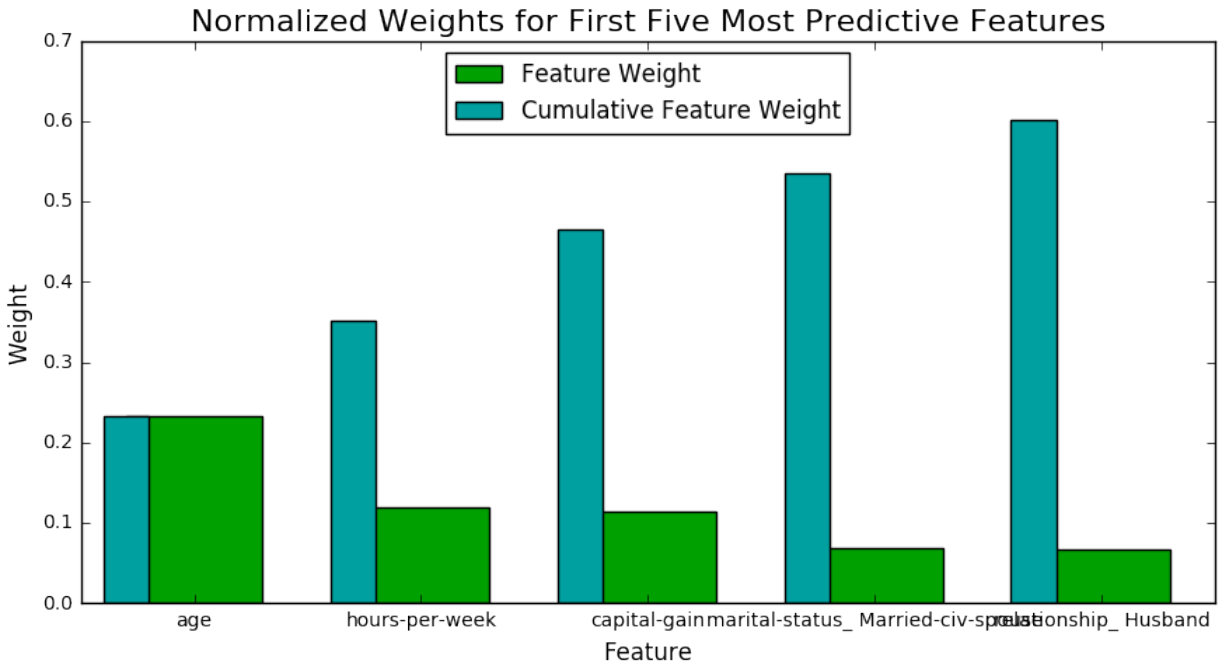
- Import a supervised learning model from sklearn if it is different from the three used earlier.
- Train the supervised model on the entire training set.
- Extract the feature importances using `'.feature_importances_'`.

In [13]:
```
# TODO: Import a supervised learning model that has 'feature_importances_'
from sklearn.ensemble import RandomForestClassifier

# TODO: Train the supervised model on the training set
model = RandomForestClassifier(n_estimators=10)
model.fit(X_train, y_train)

# TODO: Extract the feature importances
importances = model.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: There are two features (age and education-num) from what I discussed in Question 6 are present in this visualization, based on rank of importance of the features. This just affirms that age and education number are very important for predicting one's annual income level. I think the three features (hours-per-week, capital-gain, relationship_Husband) present in this visualization that are absent from my previous discussions are more relevant because it shows that the number of hours per week a person works, the amount of financial capital gain, and the status of a person in a family, are more important than the other features, which make sense. This is consistent with that the more a person works per week, the higher income the person would make, that people with more capital gains tend to have higher income, and different income levels made by different genders. Overall, this demonstrates the importance of utilizing machine learning based algorithms to objectively analyze the rank the data set for better prediction, instead of simply relying on one's intuition.

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 [14]:
```
# 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(importanc
es)[::-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, bes
t_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, red
uced_predictions, beta = 0.5))
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8494
F-score on testing data: 0.7008

Final Model trained on reduced data
------
Accuracy on testing data: 0.7940
F-score on testing data: 0.5480
```

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 reducted data using only five features are lower than the respective F-score and accuracy score when all features are used (the accuracy score is only slightly lower). If training time was a factor, I would consider using the reduced data as my training set, as the reduced data will reduce the time needed to train the model and improve efficiency. But the final decision will need to be made to balance the training time and all the measuring metrics (accuracy score, F-score). Furthermore, including all the features in the model only produce modestly higher accuracy score, but it could potentially overfit the model (with the inclusion of less important features), potentially making prediction of additional test data prone to inaccuracies.

> **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.