Finding Donors for Higher Education Foundation

1. Introduction

1.1 Background

I am a senior application developer from the Florida State University Foundation. The mission of our organization is to enhance the academic vision and priorities of FSU through its organized fundraising activities and funds management. Established in 1960, the FSU Foundation today manages an endowment of more than $509 million (as of June 30, 2019).

Like other nonprofits. we have collected and are managing over a million of records of constituent data. We implemented the web based Blackbaud CRM system over 5 years ago. which is an expansive, multifaceted constituent relationship management (CRM) software designed for nonprofits. Blackbaud CRM is a highly inclusive constituent database with a variety of features to boost the fundraising and donor stewardship efforts long-term.

With the big database in place, there’s always a gold mine of insights that add great value to organization’s business decision making. Data science has many tools in its bag such as Analytics, Deep Learning, Machine Learning, etc. to dig this mine.

Our IS management has been aware of this trend and started the Data Analytics Workgroup early this year. Our first step is to train ourselves by enrolling the IBM Data Science Professional Certificate program. The Foundation is going to reimburse us up to $300 for completion. Since we brought problems to learn, I would like to utilize the dataset related to the foundation organization and use the skills and the tools learnt from Coursera to solve the problems. This is the reason why I choose my Capstone project as a machine learning approach to fundraising success in higher education.

1.2 Problem

New donor acquisition and current donor promotion are the two major programs in fundraising for higher education and developing proper targeting strategies plays an important role in the both programs.

The size of the donor group determines the scope of fundraising. Acquiring new donors is always important for fundraisers. However, contacting randomly without a clear targeting strategy can be inefficient and may be disturbing for those who do not wish to be contacted. Keeping asking the wrong people will annoy them and give them a bad feeling about the institution. As a result, a proper targeting strategy that helps fundraisers to locate the potential donors is important not only for boosting fundraising efficiency but also for protecting the university's reputation. An efficient fundraising program should always start with developing a targeting strategy to accurately identify the pool of prospects.

The problem of targeting potential donors can be modeled as a supervised learning problem in machine learning, with the goal of identifying potential donors from all the candidates according to their personal and affiliation factors. Using machine learning methods, two important problems can be addressed in the higher education fundraising industry, identifying prospective donors and “promising donors", a term referring to donors who will upgrade their pledge. The first problem is to look for new donors from alumni, alumni families or relatives. The second problem is to look for existing donors who have a high potential to increase their donation.

1.3 Interest

Specifically in this exercise project, we will employ several supervised algorithms to accurately model individuals' income using data collected from the 1994 U.S. Census (we use the public dataset in the project instead of the dataset pulled from our proprietary data warehouse). Our 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.

2. Data Acquisition and Exploring

2.1 Data source

The dataset for this project originates from the UCI Machine Learning Repository. 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. 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.

2.2 Features

After exploring the featureset, it can be sumarized as below:

- **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.

2.2 Exploring the data

Let's start with importing the necessary libaries, reading in the data, and checking out the dataset.

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 visualisation libraries
import visuals as vs
import seaborn as sns
import matplotlib.pyplot as plt

# 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=3))
```

	age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-los
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	0.0
1	50	Self-emp-not-inc	Bachelors	13.0	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.0	0.0
2	38	Private	HS-grad	9.0	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.0	0.0

A simple investigation of the dataset can determine how many individuals fit into either group, and tell us about the percentage of these individuals making more than $50,000.

Let's take a look at 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'`.

In [2]:
```
#Checking out the datatypes of the features
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45222 entries, 0 to 45221
Data columns (total 14 columns):
age                45222 non-null int64
workclass          45222 non-null object
education_level    45222 non-null object
education-num      45222 non-null float64
marital-status     45222 non-null object
occupation         45222 non-null object
relationship       45222 non-null object
race               45222 non-null object
sex                45222 non-null object
capital-gain       45222 non-null float64
capital-loss       45222 non-null float64
hours-per-week     45222 non-null float64
native-country     45222 non-null object
income             45222 non-null object
dtypes: float64(4), int64(1), object(9)
memory usage: 4.8+ MB
```

In [3]:
```
# Total number of records
n_records = len(data)

# Number of records where individual's income is more than $50,000
n_greater_50k = len(data[data['income'] == '>50K'])

# Number of records where individual's income is at most $50,000
n_at_most_50k = len(data[data['income'] == '<=50K'])

# Percentage of individuals whose income is more than $50,000
greater_percent = 100 * n_greater_50k / 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}%".format(greater_percent))
```

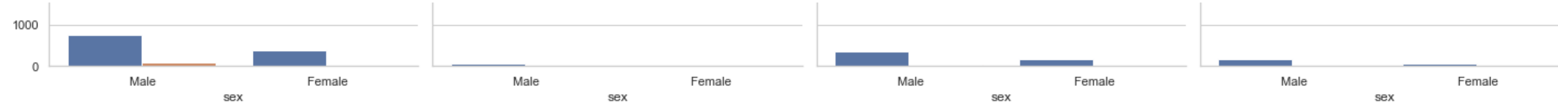
```
Total number of records: 45222
Individuals making more than $50,000: 11208
Individuals making at most $50,000: 34014
Percentage of individuals making more than $50,000: 24.78%
```

We can also visualize the relationship between different features of an individual, and their incomes.

Let's see breakdown of the counts of people earning above or below 50K based on their sex and education levels.

In [4]:
```
sns.set(style="whitegrid", color_codes=True)
sns.catplot("sex", col='education_level', data=data, hue='income', kind="count", col_wrap=4);
```


education_level = Bachelors
education_level = HS-grad
education_level = 11th
education_level = Masters
education_level = 9th
education_level = Some-college
education_level = Assoc-acdm
education_level = 7th-8th
education_level = Doctorate
education_level = Assoc-voc
education_level = Prof-school
education_level = 5th-6th
education_level = 10th
education_level = Preschool
education_level = 12th
education_level = 1st-4th
count
0
1000
2000
3000
4000
5000
6000
7000
8000
income
<=50K
>50K



3. Preparing the Data

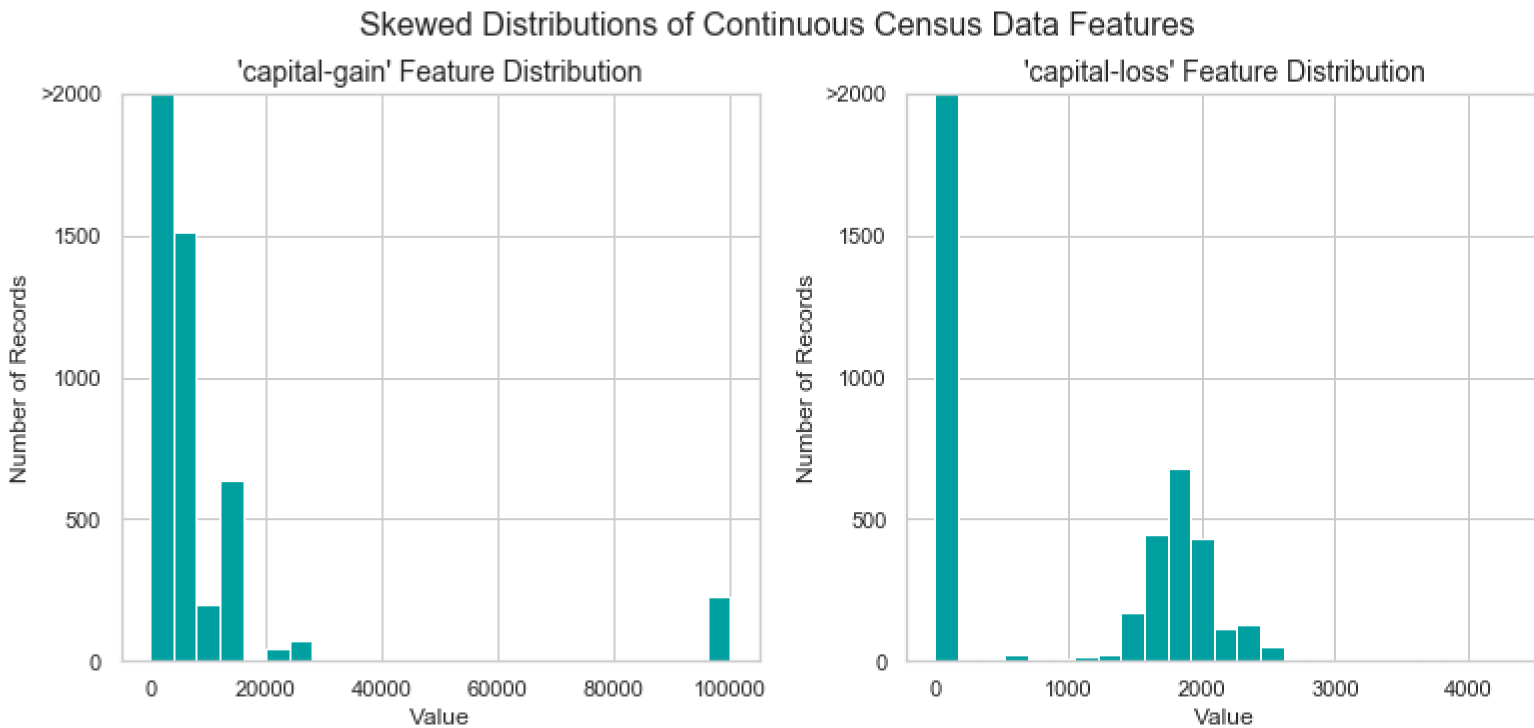
Before data can be used as input for machine learning algorithms, it often must be cleaned, formatted, and restructured — this is typically known as **preprocessing**. Fortunately, for this dataset, there are no invalid or missing entries we must deal with, however, there are some qualities about certain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

3.1 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'.

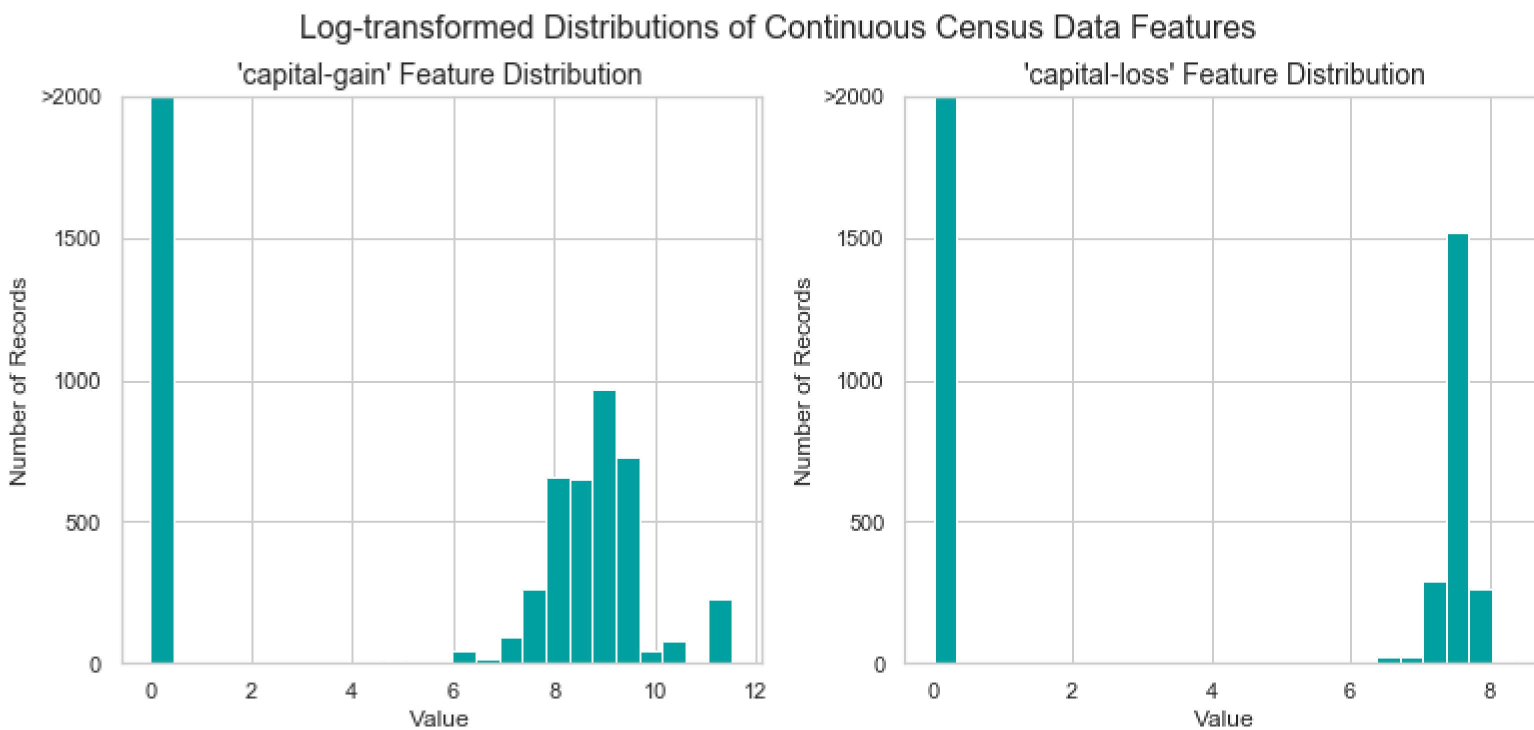
Let's plot a histogram of these two features and see how they are distributed.

In [6]:
```
# 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.

In [7]:
```
# 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)
```



3.2 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.

In [8]:
```
# 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	race	sex	capital-gain	cap
0	0.30137	State-gov	Bachelors	0.8	Never-married	Adm-clerical	Not-in-family	White	Male	0.02174	0.0

3.3 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.

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 [9]:
```
# One-hot encode the 'features_raw' data using pandas.get_dummies()
features = pd.get_dummies(features_raw)

# Encode the 'income_raw' data to numerical values
income = income_raw.apply(lambda x: 1 if x == '>50K' else 0)

# 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)
# Left uncommented due to output size
```

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

3.4 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.

In [10]:
```
# Import train_test_split
from sklearn.model_selection import train_test_split

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

4. Evaluating Model Performance

In this section, we will investigate four different algorithms, and determine which is best at modeling the data.

4.1 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:

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

Fβ=(1+β2)·precision·recall(β2·precision)+recall

In particular, when $\beta = 0.5$ β=0.5, more emphasis is placed on precision.

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, we would identify no one as donors.

4.2 Naive Predictor Performace

What 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?

In [11]:
```
TP = np.sum(income) # Counting the ones as this is the naive case.
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

# Calculate accuracy, precision and recall
accuracy = TP/income.count()
recall = TP/(TP + FN)
precision = TP/(TP + FP)

# Calculate F-score using the formula above for beta = 0.5 and correct values for precision and recall.
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(accuracy, fscore))
```

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

5. Supervised Learning Models

5.1 Model Application

Now we'll pick three supervised learning models above that are appropriate for this problem, and test them on the census data.

Decision Trees

- Real world application: Decision Trees and, in general, CART (Classification and Regression Trees) are often used in financial analysis. A concrete example of it is: for predicting which stocks to buy based on past peformance. Reference
- Strengths:
  - Able to handle categorical and numerical data.
  - Doesn't require much data pre-processing, and can handle data which hasn't been normalized, or encoded for Machine Learning Suitability.
  - Simple to understand and interpret.
- Weaknesses:
  - Complex Decision Trees do not generalize well to the data and can result in overfitting.
  - Unstable, as small variations in the data can result in a different decision tree. Hence they are usually used in an ensemble (like Random Forests) to build robustness.
  - Can create biased trees if some classes dominate.
- Candidacy: Since a decision tree can handle both numerical and categorical data, it's a good candidate for our case (although, the pre-processing steps might already mitigate whatever advantage we would have had). It's also easy to interpret, so we will know what happens under the hood to interpret the results.

Support Vector Machines (SVM)

- Real world application: Example of a real world use of SVMs include image classification and image segmentation. For example: Face detection in an image. Reference
- Strenghs:
  - Effective in high dimensional spaces, or when there are a lot of features.
  - Kernel functions can be used to adapt to different cases, and can be completely customized if needed. Thus SVMs are versatile.
- Weaknesses:
  - Doesn't perform well with large datasets.
  - Doesn't directly provide probability estimates.
- Candidacy: SVMs were chosen because of their effectiveness given high dimensionality. After incorporating dummy variables, we have more than 100 features in our dataset, so SVMs should be a classifier that works regardless of that. Also, our dataset is not that large to be a deterrent.

Ensemble methods: AdaBoost

- Real world application: Ensemble methods are used extensively in Kaggle competitions, usually in image detection. A real world example of Adaboost is object detection in image, ex: identifying players during a game of basketball. Reference
- Strength:
  - Ensemble methods, including Adaboost are more robust than single estimators, have improved generalizability.
  - Simple models can be combined to build a complex model, which is computationally fast.
- Weaknesses:
  - If we have a biased underlying classifier, it will lead to a biased boosted model.
- Candidacy: Ensemble methods are considered to be high quality classifiers, and adaboost is the one of most popular boosting algorithms. We also have a class imbalance in our dataset, which boosting might be robust to.

5.2 Creating a Training and Predicting Pipeline

To properly evaluate the performance of each model we'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.

In [12]:
```
# Import two metrics from sklearn - fbeta_score and accuracy_score

from sklearn.metrics import fbeta_score, accuracy_score

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

    results = {}

    # 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

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

    # 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

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

    # Compute accuracy on the first 300 training samples
    results['acc_train'] = accuracy_score(y_train[:300],predictions_train)

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

    # Compute F-score on the the first 300 training samples
```

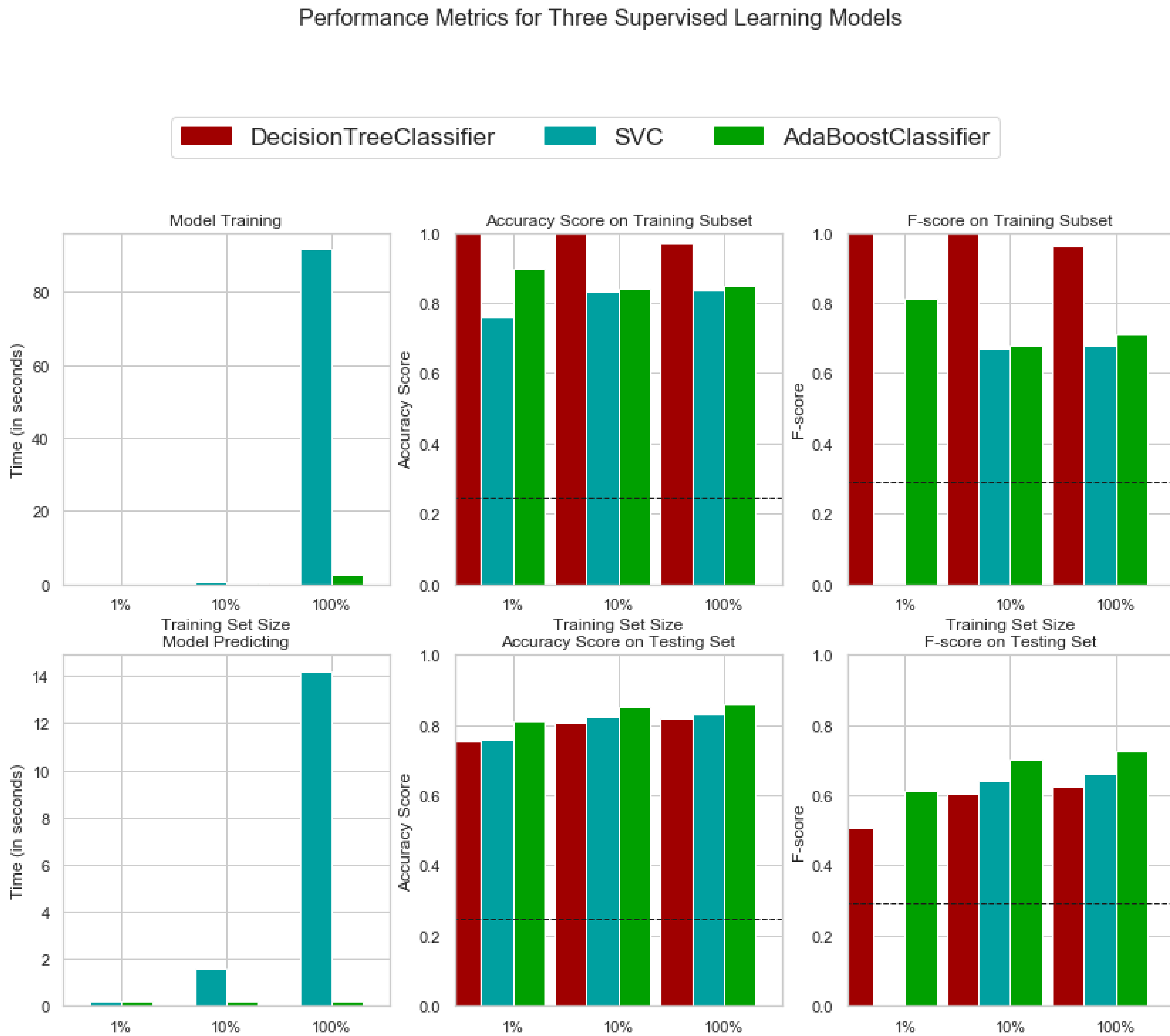
```
    results['f_train'] = fbeta_score(y_train[:300],predictions_train,0.5)

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

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

    # Return the results
    return results
```

5.3 Model Evaluation

Let's train and test the models on training sets of different sizes to see how it affects their runtime and predictive performance (both on the test, and training sets).

In [13]:
```
# Import the three supervised learning models from sklearn
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier

# Initialize the three models, the random states are set to 101 so we know how to reproduce the model later
clf_A = DecisionTreeClassifier(random_state=101)
clf_B = SVC(random_state = 101, gamma='auto')
clf_C = AdaBoostClassifier(random_state = 101)

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

# Collect results on the learners
results = {}
for clf in [clf_A, clf_B, clf_C]:
    clf_name = clf.__class__.__name__
    results[clf_name] = {}
    for i, samples in enumerate([samples_1, samples_10, samples_100]):
        results[clf_name][i] = \
        train_predict(clf, samples, X_train, y_train, X_test, y_test)

# Run metrics visualization for the three supervised learning models chosen
vs.evaluate(results, accuracy, fscore)
```

```
DecisionTreeClassifier trained on 362 samples.
DecisionTreeClassifier trained on 3618 samples.
DecisionTreeClassifier trained on 36177 samples.

C:\Users\tjiang\AppData\Local\Continuum\anaconda3\lib\site-packages\sklearn\metrics\classification.py:1437: U
ndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to no predicted samples.
  'precision', 'predicted', average, warn_for)

SVC trained on 362 samples.
SVC trained on 3618 samples.
SVC trained on 36177 samples.
AdaBoostClassifier trained on 362 samples.
AdaBoostClassifier trained on 3618 samples.
AdaBoostClassifier trained on 36177 samples.

C:\Users\tjiang\Tiehu\Online Courses\IBM Data Science Professional Certificate\Capstone project\visuals.py:11
8: UserWarning: Tight layout not applied. tight_layout cannot make axes width small enough to accommodate all
axes decorations
  pl.tight_layout()
```


Performance Metrics for Three Supervised Learning Models
DecisionTreeClassifier
SVC
AdaBoostClassifier
Model Training
Time (in seconds)
Training Set Size
Accuracy Score on Training Subset
Accuracy Score
F-score on Training Subset
F-score
Model Predicting
Accuracy Score on Testing Set
F-score on Testing Set
1%
10%
100%

Training Set Size Training Set Size Training Set Size

We can also print out the values in the visualizations above to examine the results in more detail.

In [14]:
```
#Printing out the values
for i in results.items():
    print(i[0])
    display(pd.DataFrame(i[1]).rename(columns={0:'1%', 1:'10%', 2:'100%'}))
```

DecisionTreeClassifier

	1%	10%	100%
acc_test	0.754008	0.806081	0.817026
acc_train	1.000000	0.996667	0.970000
f_test	0.505562	0.602673	0.624711
f_train	1.000000	0.997191	0.963855
pred_time	0.008007	0.013011	0.009009
train_time	0.003003	0.027021	0.402366

SVC

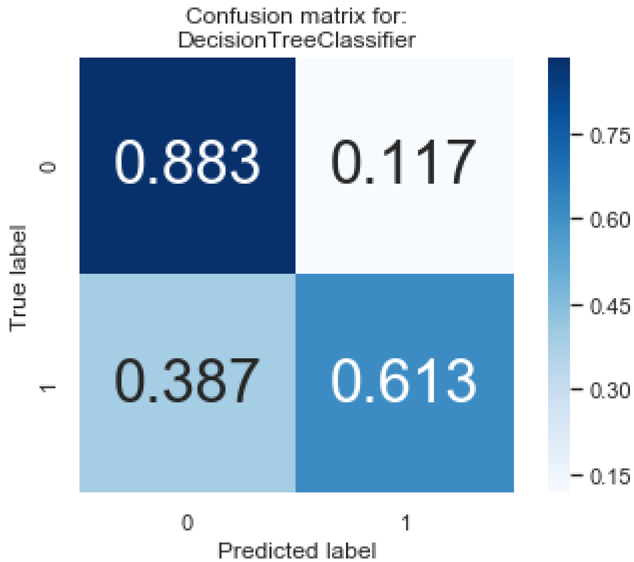
	1%	10%	100%
acc_test	0.756219	0.822443	0.830072
acc_train	0.760000	0.833333	0.836667
f_test	0.000000	0.640209	0.659238
f_train	0.000000	0.669643	0.677966
pred_time	0.179146	1.574323	14.203584
train_time	0.008532	0.722137	91.605304

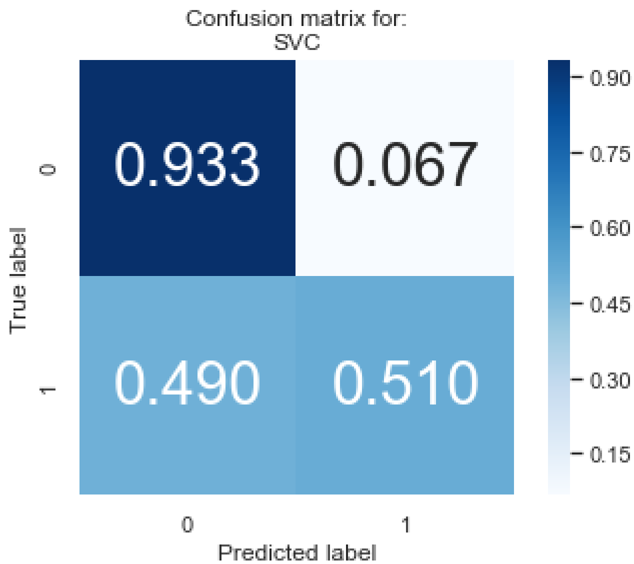
AdaBoostClassifier

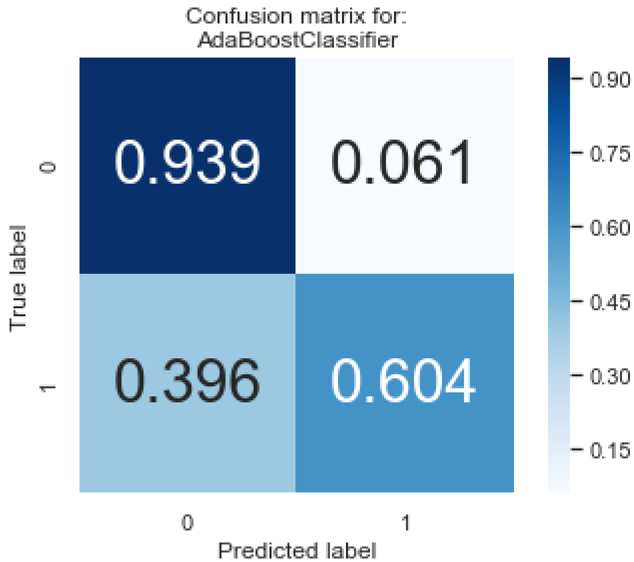
	1%	10%	100%
acc_test	0.810392	0.849862	0.857601
acc_train	0.896667	0.840000	0.850000
f_test	0.610473	0.701882	0.724551
f_train	0.811688	0.680147	0.711538
pred_time	0.185151	0.183650	0.189656
train_time	0.062551	0.269720	2.542072

And visualize the confusion matrix for the results.

In [15]:
```
#Visualizing the confusion matrix for each classifier
from sklearn.metrics import confusion_matrix

for i,model in enumerate([clf_A,clf_B,clf_C]):
    cm = confusion_matrix(y_test, model.predict(X_test))
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] # normalize the data

    # view with a heatmap
    plt.figure(i)
    sns.heatmap(cm, annot=True, annot_kws={"size":30},
            cmap='Blues', square=True, fmt='.3f')
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title('Confusion matrix for:\n{}'.format(model.__class__.__name__));
```


Confusion matrix for:
DecisionTreeClassifier
0.883
0.117
0.387
0.613
True label
0
1
Predicted label
0
1
0.75
0.60
0.45
0.30
0.15


Confusion matrix for:
SVC
0.933
0.067
0.490
0.510
True label
0
1
Predicted label
0
1
0.90
0.75
0.60
0.45
0.30
0.15



Looking at the results above, out of the three models, AdaBoost is the most appropriate for our task.

First and foremost, it is the classifier that performs the best on the testing data, in terms of both the *accuracy* and *f-score*. It also takes resonably low time to train on the full dataset, which is just a fraction of the 120 seconds taken by SVM, the next best classifier to train on the full training set. So it should scale well even if we have more data.

By default, Adaboost uses a decision stump i.e. a decision tree of depth 1 as its base classifier, which can handle categorical and numerical data. Weak learners are relatively faster to train, so the dataset size is not a problem for the algorithm.

How does Adaboost work?

1. Adaboost works by combining several simple learners (for ex: decision trees), to create an ensemble of learners that can predict whether an individual earns above 50k or not.
2. Each of the learners, in our case decision trees, are created using "features" we have about individuals (eg. age, occupation, education, etc) create a set of rules that can predict a person's income.
3. During the training process, which lasts for several rounds, the Adaboost algorithm looks at instances where it has predicted badly, and prioritizes the correct prediction of those instances in the next round of raining.
4. With each round, the model finds the best learner (or decision tree) to incorporate into the ensemble, repeating the process for the specified number of rounds, or till we can't improve the predictions further.
5. All the learners are then combined to make a final ensembled model, where they each vote to predict if a person earns more than 50k or not. Usually we take the majority of the votes to make a final prediction.
6. Using this model with the census information of individuals, we can predict the same information for a potential new donor and predict if they earn more than 50K or not, and thus make a decision on the likeliness of them donating to charity.

6. Improving our Model: Model Tuning

Using grid search (`GridSearchCV`) with different parameter/value combinations, we can tune our model for even better results.

For Adaboost, we'll tune the n_estimators and learning rate parameters, and also the base classifier paramters (remember our base classifier for the Adaboost ensemble is a Decision tree!).

In [16]:
```
# Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

# Initialize the classifier
clf = AdaBoostClassifier(base_estimator=DecisionTreeClassifier())

# Create the parameters list you wish to tune
parameters = {'n_estimators':[50, 120],
              'learning_rate':[0.1, 0.5, 1.],
              'base_estimator__min_samples_split' : np.arange(2, 8, 2),
              'base_estimator__max_depth' : np.arange(1, 4, 1)
             }

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

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

# 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)))
print(best_clf)
```

```
C:\Users\tjiang\AppData\Local\Continuum\anaconda3\lib\site-packages\sklearn\model_selection\_split.py:1978: F
utureWarning: The default value of cv will change from 3 to 5 in version 0.22. Specify it explicitly to silen
ce this warning.
  warnings.warn(CV_WARNING, FutureWarning)

Unoptimized model
------
Accuracy score on testing data: 0.8342
F-score on testing data: 0.6619

Optimized Model
------
Final accuracy score on the testing data: 0.8703
Final F-score on the testing data: 0.7529
AdaBoostClassifier(algorithm='SAMME.R',
                   base_estimator=DecisionTreeClassifier(class_weight=None,
                                                         criterion='gini',
                                                         max_depth=3,
                                                         max_features=None,
                                                         max_leaf_nodes=None,
                                                         min_impurity_decrease=0.0,
                                                         min_impurity_split=None,
                                                         min_samples_leaf=1,
                                                         min_samples_split=6,
                                                         min_weight_fraction_leaf=0.0,
                                                         presort=False,
                                                         random_state=None,
                                                         splitter='best'),
                   learning_rate=0.5, n_estimators=50, random_state=None)
```

6.1 Final Model Evaluation

Results:

Metric	Benchmark Predictor	Unoptimized Model	Optimized Model
Accuracy Score	0.2478	0.8186	0.8702
F-score	0.2917	0.6282	0.7526

The optimized model has an accuracy of 0.8702 and F-score of 0.7526.

These scores are better than the umpotimized model, while being substantially better than the benchmark predictor.

6.2 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.

Feature Relevance Observation

We know that there are thirteen available features for each individual on record in the census data. Of these thirteen records, we can guess which five features might be most important for prediction. Let's give that a go.

In my opinion, are most important for prediction are:

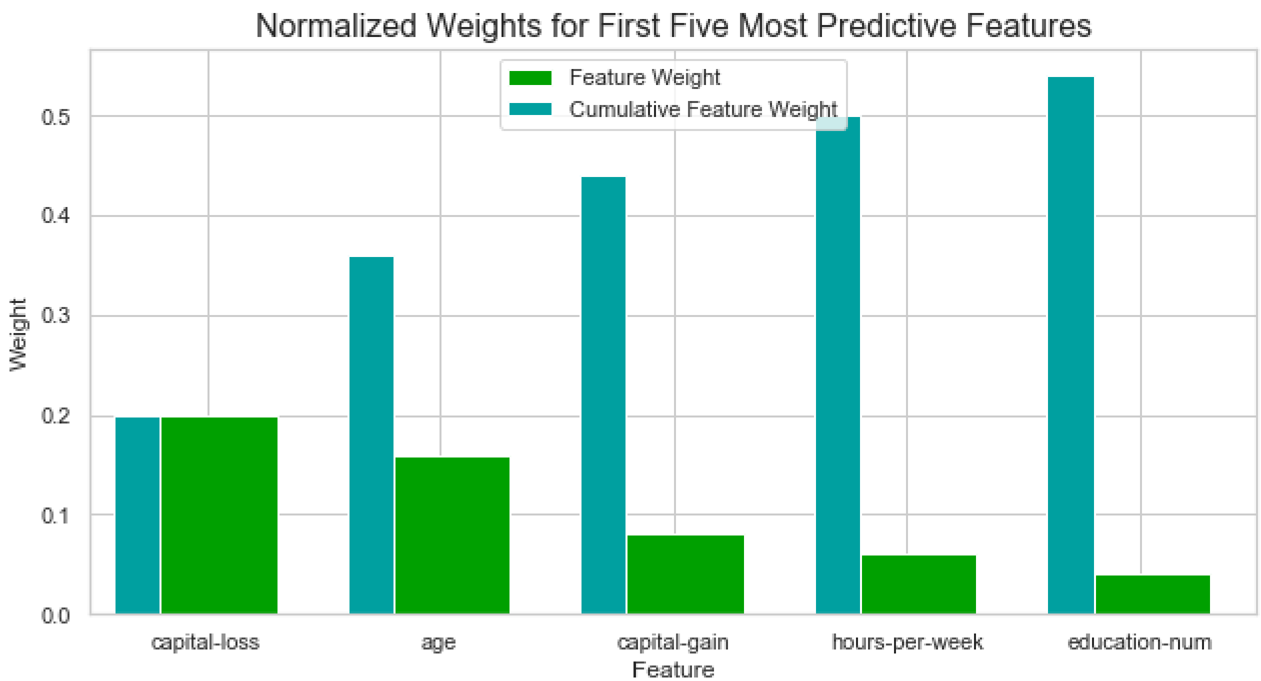
1. occupation: Different jobs have different payscales. Some jobs pay higher than others.
2. education: People who have completed a higher level of education are better equipped to handle more technical/specialized jobs that pay well.
3. age: As people get older, they accumulate greater weatlh.
4. workclass: The working class they belong to can also be correlated with how much money they make.
5. hours-per-week: If you work more hours per week, you're likely to earn more.

These are all ranked according the the impact I believe they have on a person's income. Occupation's ranked number one as different jobs have different payscales. People with higher education are more likely to earn better.

Now let's see how our algorithm ranks the features acc. to their importance.

Extracting Feature Importance

In [17]:
```
# Import a supervised learning model that has 'feature_importances_'

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

# Extract the feature importances
importances = model.feature_importances_

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



Of the five features predicted in the earlier section, 3 of them, Age, hours per week, education-num (which is a numerical label for education) are included in the list of features considered most important by Adaboost, although with different rankings.

I didn't consider two important features, capital-gain and capital-loss, partly due to my lack of understanding of what they meant. After researching what they mean (profit or loss from on the sale of assets/property), it makes sense for these features to be important. People who have earned profits from sale of assets are definitely likelier to earn higher, while those who incurred losses are likely to have had lower total income.

Feature Selection

An interesting question here is, 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.

Let's see how a model that is trained only on the selected features, performs.

In [18]:
```
# Import functionality for cloning a model
from sklearn.base import clone

# Reduce the feature space
X_train_reduced = X_train[X_train.columns.values[(np.argsort(importances)[::-1])[:5]]]
X_test_reduced = X_test[X_test.columns.values[(np.argsort(importances)[::-1])[:5]]]

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

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

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

```
Final Model trained on full data
------
Accuracy on testing data: 0.8703
F-score on testing data: 0.7529

Final Model trained on reduced data
------
Accuracy on testing data: 0.8437
F-score on testing data: 0.7065
```

7. Conclusion

On a reduced dataset, the final model's accuracy and f-score are still very comparable to the results on the full dataset.

The acccuracy is ~2.7% lower, while the f-score is ~5% lower. Even though Adaboost is relatively faster than one of the other classifiers that we tried out, I'd still consider training on the reduced data (acc. to features) if training time was a factor, and we have more training points to process. This decision will also depend on how important accuracy and f-scores are (or if f-score is more important than the accuracy, as the dip for that is larger than the dip in accuracy), to make a final decision regarding this.