Supervised Learning

Finding Donors for *CharityML*

Author: Qi Meng

Using individuals' data collected from 1994 U.S. Census, we are going to construct a model that accurately predicts whether an individual makes more than $50,000. 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*".

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 [9]:
```
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 visuals as vs
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')

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

display(data.head(n=1))
```

	age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	0.0	40.0	United-States

Implementation: Data Exploration

Finding the percentage of these individuals making more than $50,000.

In [10]:
```
n_records = data.index.shape[0]
n_greater_50k = data[data['income'] == '>50K'].shape[0]
n_at_most_50k = data[data['income'] == '<=50K'].shape[0]

greater_percent = data[data['income'] == '>50K'].shape[0]/data.shape[0] * 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(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: 24.78439697492371%
```

Featureset Exploration

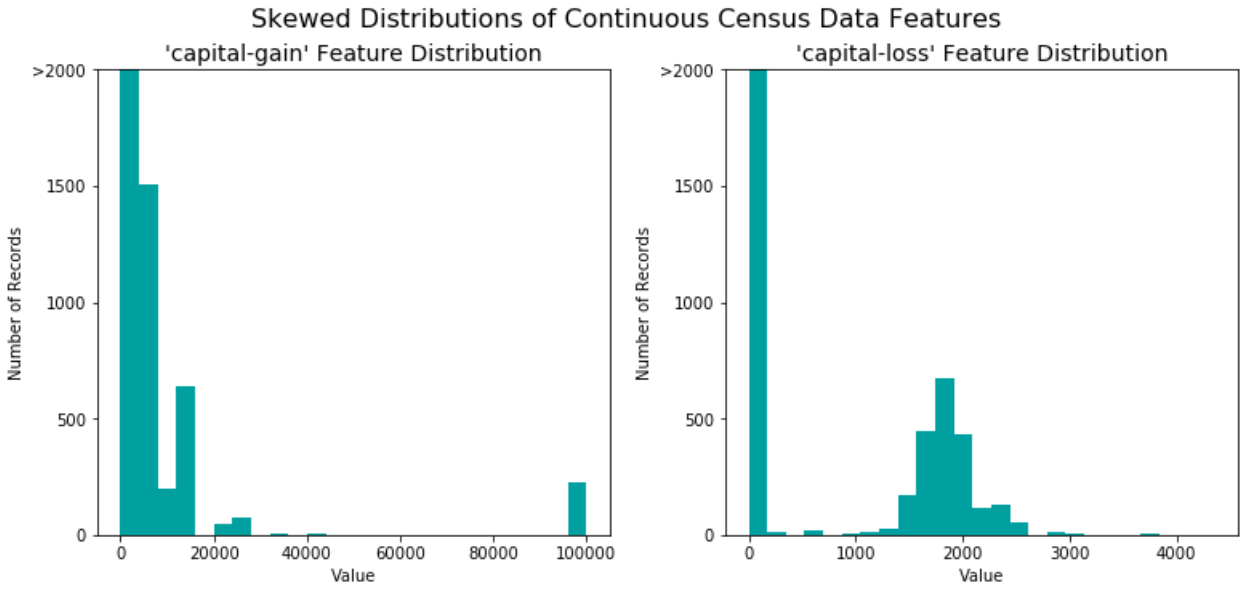
- **age**: continuous.
- **workclass**: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
- **education**: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
- **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

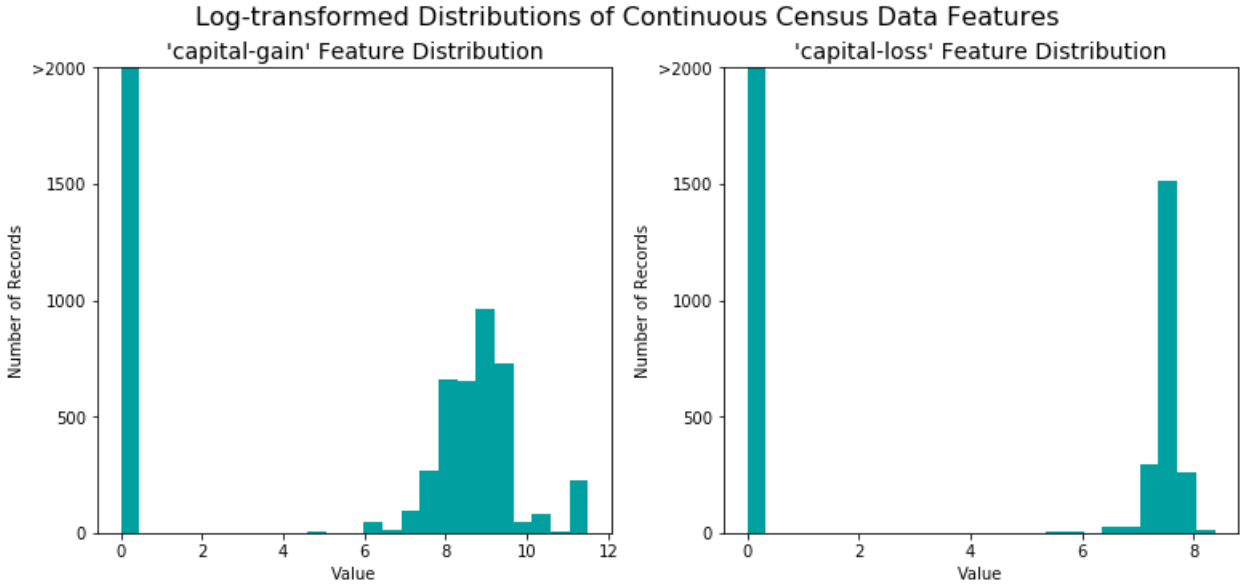
Transforming Skewed Continuous Features

Algorithms can be sensitive to 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'`.

In [11]:
```
income_raw = data['income']
features_raw = data.drop('income', axis = 1)

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



In [12]:
```
# 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.log(x + 1))

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



Normalizing Numerical Features

In [ ]:
```
```

In [13]:
```
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-per-week']

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

display(features_log_minmax_transform.head(n = 5))
```

	age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours per wee
0	0.301370	State-gov	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family	White	Male	0.667492	0.0	0.39795
1	0.452055	Self-emp-not-inc	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.000000	0.0	0.12244
2	0.287671	Private	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.000000	0.0	0.39795
3	0.493151	Private	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.000000	0.0	0.39795
4	0.150685	Private	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.000000	0.0	0.39795

Implementation: Data Preprocessing

From exploring the data, we can see several features for each record are non-numeric, therefore, we are applying the one-hot encoding scheme.

In [14]:
```
features_final = pd.get_dummies(features_log_minmax_transform)

income = income_raw.apply(lambda x: 0 if x == '<=50K' else 1)

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

# print encoded
```

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

Shuffle and Split Data

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 [15]:
```
from sklearn.cross_validation import train_test_split

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 [ ]:
```

```

In [17]:
```
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import fbeta_score

# create the income predition and set all prediction results to 1, income >50K

'''
TP = np.sum(income) # Counting the ones as this is the naive case. Note that 'income' is the 'incc
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
'''
TP = np.sum(income)
FP = income.count() - TP

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

accuracy = TP/FP
recall = TP/(TP + FN)

precision = TP/(TP + FP)


beta=0.5
fscore = (1+beta**2)*(accuracy*recall)/(beta**2*accuracy+recall)

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

```
Naive Predictor: [Accuracy score: 0.3295, F-score: 0.3805]
```

In [ ]:
```
```

Supervised Learning Models

Model Application

Following are the three supervised learning models will be used in this case

Logistic Regression

Logistic regression is being widely used for binary classification (ref: https://towardsdatascience.com/the-logistic-regression-algorithm-75fe48e21cfa (https://towardsdatascience.com/the-logistic-regression-algorithm-75fe48e21cfa)). Instead of giving a continuous outcome like linear regression model, logistic regression model will generate a discreate outcome. One of the biggest advantages is that logistic regression does not need too many computational resources. In addition, logistic regression does not require the input features to be scaled. However, logistic regression could not solve linear regression problems. The logistic regression could be a good candidate for this problem, is because this is a binary classification problem, to find out whether an individual makes more than $50, 000, and all the input variables are binary variables.

Random Forest

Random Forest method is also widely used for classification regression problems. It was widely used in banking, medicine, and e-commerce (http://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/ (http://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/)) in finding loyal customers, and also helpful in identifying the disease. Like its name, Random and Forest are the two biggest attributes of this method. The method created decision tree predictions randomly based on the training data to form a decision tree ‘forest’, and the final prediction result will depend on the majority voting from all the decision tree results. The advantage of Random Forest method is that it can avoid overfitting, and the disadvantage is that it might take a lot of memory to run. It is a good candidate for this model, as it can improve the result by preventing overfitting the model.

Gradient Boosting

Gradient Boosting used in supervised learning to solve regression and classification problems. Gradient Boosting can be used in predicting house prices, also ranking (https://en.wikipedia.org/wiki/Gradient_boosting (https://en.wikipedia.org/wiki/Gradient_boosting)). The advantage of Gradient Boosting that it is easy to implement, and also strong in building a strong model as it improves after each training. The disadvantage is that it takes a lot of memory and could be overfitting for small datasets. Gradient Boosting is a good candidate here, as it provides strong predictions for large datasets.

Implementation - Creating a Training and Predicting Pipeline

In [18]:
```
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 = {}

    start = time() # Get start time
    learner.fit(X_train[:sample_size],y_train[:sample_size])
    end = time() # Get end time

    results['train_time'] = end - start

    start = time() # Get start time
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train[:300])
    end = time() # Get end time

    results['pred_time'] = end - start

    results['acc_train'] = accuracy_score(y_train[:300], predictions_train)

    results['acc_test'] = accuracy_score(y_test, predictions_test)

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

    # 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__, sample_size))

    # Return the results
    return results
```

In [19]:
```
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.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,beta=0.5)

    # 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
```

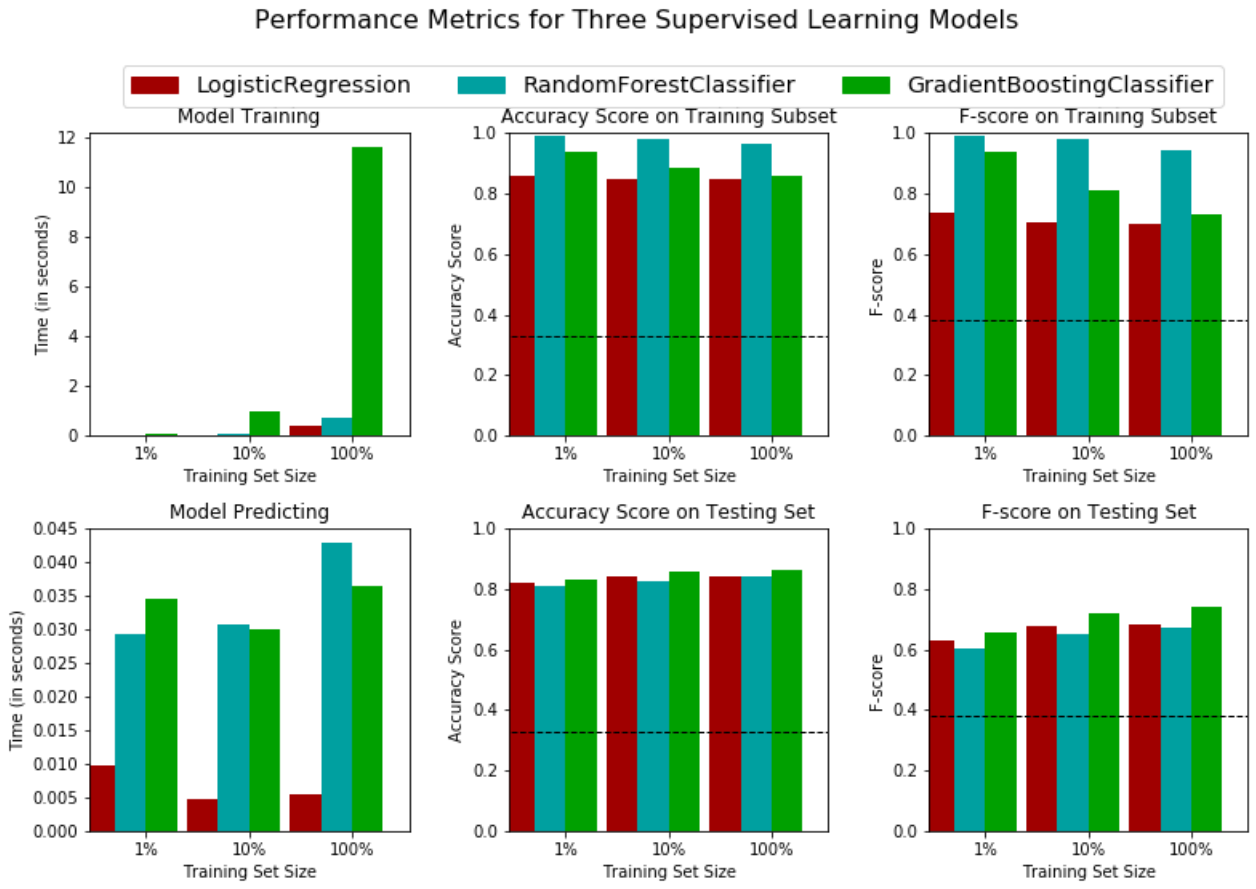
Implementation: Initial Model Evaluation

- Import the three supervised learning models
- 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.
- 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.

In [ ]:
```
```

In [20]:
```
# Import the three supervised learning models from sklearn

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier

# Initialize the three models
clf_A = LogisticRegression(random_state= 23)
clf_B = RandomForestClassifier(random_state= 23)
clf_C = GradientBoostingClassifier(random_state= 23)


# Calculate the number of samples for 1%, 10%, and 100% of the training data
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)

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

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



In [ ]:

Improving Results

We are going to 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.

Choosing the Best Model

Among the three models, Gradient Boosting is the best model.

1. Gradient Boosting model has the highest accuracy score as well as F-score for the all testing datasets, and also relatively high accuracy score for the training dataset.
2. Gradient Boosting also have relatively low run time for training and predicting, while Random Forest model has the highest run time among the other two for the 100% testing dataset.
3. In general, Gradient Boosting will be a good fit for the model selection, as it has low run time but also high accuracy and recall score.

Intuitive Explanation

Gradient Boosting method is using multiple ‘weak learners’ (decision trees) to classify the data, and for the next time, for the following rounds, weak learner will be trained on error residuals. Eventually, we will have a strong learner that has the least error residuals.

In this case, we are trying to predict if the donor earns more than 50k. The Gradient Boosting methods starts with many decision trees to decide if the donor earns more than 50k in the training set. For the following round of training, Gradient Boosting will train its weak learner on data points that have error residuals. After repeating this procedure for many times, we will have one strong learner that gives us least error residuals, and thus the preferred model to decide if the donor earns more than 50k or not.

Implementation: Model Tuning

In [21]:
```
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import make_scorer, r2_score, fbeta_score

# Initialize the classifier
clf = GradientBoostingClassifier(random_state= 23)

# Create the parameters list you wish to tune, using a dictionary if needed.
parameters = {#'loss' : ['deviance', 'exponential'],
#               'max_depth': [1, 5],
              'n_estimators': [10,100,500],
            'learning_rate': [0.1, 1.0, 2.0]
            #,'min_samples_split': [2, 4]
             }


scorer = make_scorer(fbeta_score, beta = 0.5)

grid_obj = GridSearchCV(clf, parameters, scoring= scorer)

grid_fit = grid_obj.fit(X_train, y_train)

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_predic
print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, bet
```

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

Optimized Model
------
Final accuracy score on the testing data: 0.8719
Final F-score on the testing data: 0.7547
```

Final Model Evaluation

Results:

Metric	Unoptimized Model	Optimized Model
Accuracy Score	0.8630	0.8719
F-score	0.7395	0.7547

After optimization, the Accuracy Score improved from 0.8630 to 0.8719, and F score increased from 0.7395 to 0.7547.

naïve predictor : [Accuracy score: 0.3295, F-score: 0.3805]

Comparing to naïve predictor at the beginning, the Accuracy score and F score increased tremendously, and thus the prediction will be more robust.

Feature Importance

In [ ]:

Feature Relevance Observation

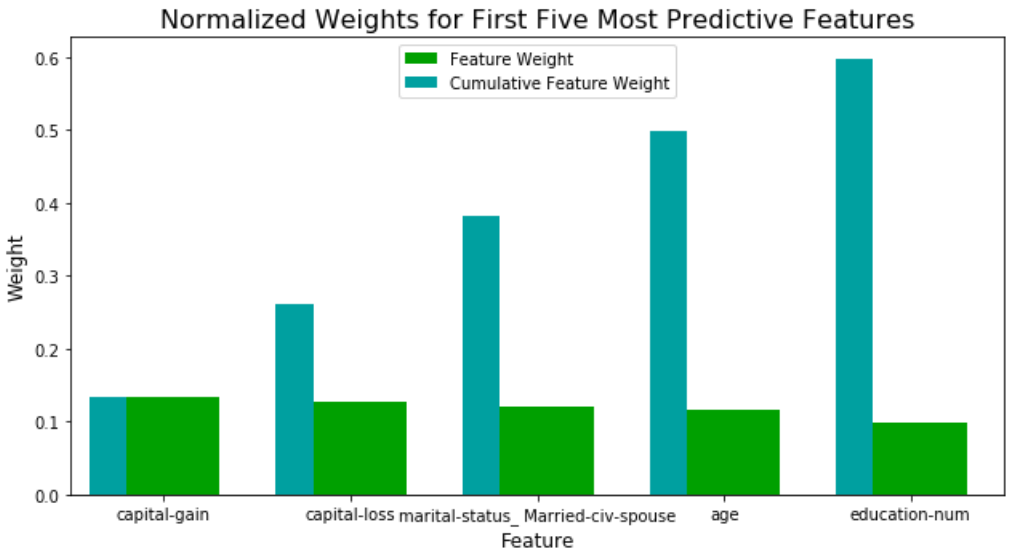
Without analyzing the data, I have picked top 5 features that I think matters the most for the model prediction result.

Top 5 variables.

1. Education. I believe the education should affect the most in terms of the income level, the higher the education level could potentially have a higher starting point for salaries.
2. Capital_gain. Capital gain is also import, as higher the capital gain, the person will have more money to manage, or invest, and could potentially affect the final income.
3. Capital_loss: Contrary to capital gain, huge capital loss could also possibly indicate higher income for the individual, at least, lower income people usually would not have large amount of capital loss.
4. Occupation. Similar to education, certain occupations, for example, Exec-managerial position such as managers or CEO will have a higher possibility of greater income.
5. Workclass. Work class means a lot for predicting the income as it already indicate the working class of the individual and each working class will have income standards to follow, and therefore, it could help the model to predict.

Implementation - Extracting Feature Importance

In [22]:
```
model = GradientBoostingClassifier().fit(X_train, y_train)

importances = model.feature_importances_

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



Comparing to the most predictive features, age and marital status are the two variables that different from my selection. Education is the most important feature the same as my top 1 selection. However, capital gain and loss was not quite important comparing to age and marital status. Age, as the second highest cumulative feature weight among the five, in my perspective, could because people who is older could potentially acquire more capital than younger people and therefore have more potential in earning higher income.