finding_donors

July 8, 2018

0.1 Supervised Learning

0.2 Project: Finding Donors for *CharityML*

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

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

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

0.3 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. 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.

0.4 Exploring the Data

The census dataset is explored by loading necessary Python libraries and 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 [2]: # 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  \
0   39   State-gov       Bachelors           13.0   Never-married

      occupation    relationship    race    sex  capital-gain  capital-loss  \
0   Adm-clerical   Not-in-family   White   Male        2174.0           0.0

   hours-per-week  native-country income
0            40.0   United-States  <=50K
```

0.4.1 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 [3]:
```
# TODO: Total number of records
n_records = len(data)

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

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

# TODO: Percentage of individuals whose income is more than $50,000
greater_percent = n_greater_50k/float(n_records) * 100.00


# 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_perce
```

```
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%
```

** 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.

0.5 Preparing the Data

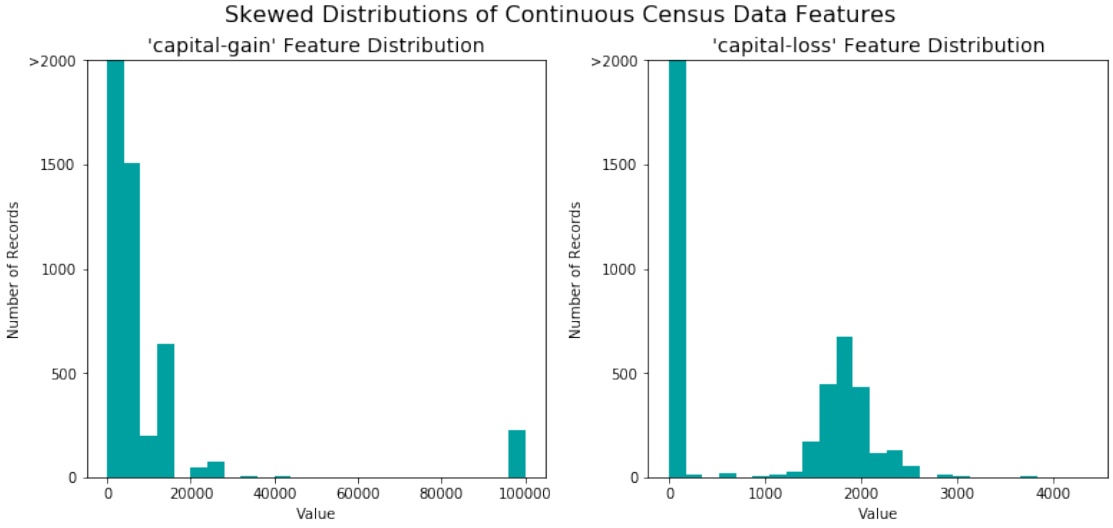
Before data can be used as input for machine learning algorithms, it often must be cleaned, formatted, and restructured — this is typically known as **preprocessing**. Fortunately, for this dataset, there are no invalid or missing entries we must deal with, however, there are some qualities about certain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

0.5.1 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'`.

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.

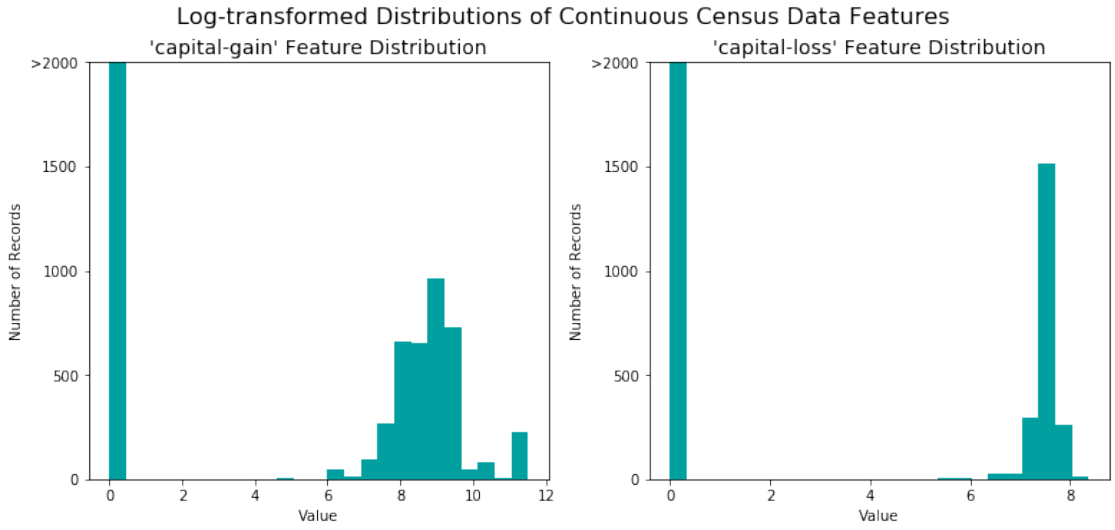
```
In [4]: # Split the data into features and target label
        income_raw = data['income']
        features_raw = data.drop('income', axis = 1)

        # 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 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 [5]:
```
# 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)
```



0.5.2 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.

Run the code cell below to normalize each numerical feature. We will use `sklearn.preprocessing.MinMaxScaler` for this.

In [6]:
```
# Import sklearn.preprocessing.StandardScaler
from sklearn.preprocessing import MinMaxScaler
```

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

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

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

```
        age          workclass education_level  education-num  \
0  0.301370          State-gov       Bachelors       0.800000
1  0.452055   Self-emp-not-inc       Bachelors       0.800000
2  0.287671            Private         HS-grad       0.533333
3  0.493151            Private            11th       0.400000
4  0.150685            Private       Bachelors       0.800000

        marital-status          occupation    relationship    race      sex  \
0        Never-married        Adm-clerical   Not-in-family   White     Male
1   Married-civ-spouse     Exec-managerial         Husband   White     Male
2             Divorced   Handlers-cleaners   Not-in-family   White     Male
3   Married-civ-spouse   Handlers-cleaners         Husband   Black     Male
4   Married-civ-spouse      Prof-specialty            Wife   Black   Female

   capital-gain  capital-loss  hours-per-week  native-country
0      0.667492           0.0        0.397959   United-States
1      0.000000           0.0        0.122449   United-States
2      0.000000           0.0        0.397959   United-States
3      0.000000           0.0        0.397959   United-States
4      0.000000           0.0        0.397959            Cuba
```

0.5.3 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() to perform one-hot encoding on the 'features_log_minmax_transform' data. - Convert the target label 'income_raw' to numerical entries. - Set records with "<=50K" to 0 and records with ">50K" to 1.

```
In [7]: # TODO: One-hot encode the 'features_log_minmax_transform' data using pandas.get_dummies
        features_final = pd.get_dummies(features_log_minmax_transform)

        # TODO: 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_final.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.
```

0.5.4 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.

Run the code cell below to perform this split.

```
In [8]: # Import train_test_split
        from sklearn.cross_validation import train_test_split

        # Split the 'features' and 'income' data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(features_final,
                                                            income,
                                                            test_size = 0.2,
                                                            random_state = 0)

        # Show the results of the split
        print("Training set has {} samples.".format(X_train.shape[0]))
        print("Testing set has {} samples.".format(X_test.shape[0]))
```

```
Training set has 36177 samples.
Testing set has 9045 samples.
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/cross_validation.py:41: DeprecationWarning: This
  "This module will be removed in 0.20.", DeprecationWarning)
```

0.6 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*.

0.6.1 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:

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

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

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

Note: Recap of accuracy, precision, recall ** Accuracy ** measures how often the classifier makes the correct prediction. It's the ratio of the number of correct predictions to the total number of predictions (the number of test data points).

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

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

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

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

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

0.6.2 Question 1 - Naive Predictor Performace

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

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

** HINT: **

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

In [9]:
```
'''
TP = np.sum(income) # Counting the ones as this is the naive case. Note that 'income' is
encoded to numerical values done in the data preprocessing step.
FP = income.count() - TP # Specific to the naive case

TN = 0 # No predicted negatives in the naive case
FN = 0 # No predicted negatives in the naive case
'''
# TODO: Calculate accuracy, precision and recall
TP = np.sum(income)
FP = income.count() - TP
TN = 0
FN = 0
accuracy = float(TP)/(TP+FP+TN+FN)
recall = float(TP)/(TP+FN)
precision = float(TP)/(TP+FP)
```

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

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

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

0.6.3 Supervised Learning Models

The following are some of the supervised learning models that are currently available in scikit-learn **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

0.6.4 Question 2 - Model Application

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

- Describe one real-world application in industry where the model can be applied.
- What are the strengths of the model; when does it perform well?
- What are the weaknesses of the model; when does it perform poorly?
- What makes this model a good candidate for the problem, given what you know about the data?

** HINT: **
Structure your answer in the same format as aboveˆ, with 4 parts for each of the three models you pick. Please include references with your answer.

Answer: For this project, I have selected two ensemble methods namely AdaBoost and Random Forest and Gaussian Naive Bayes (GaussianNB).

** Random Forest ** - **Describe one real-world application in industry where the model can be applied.** Random Forest have many real-world applications. It can be used in medicine field, where it identifies the correct combination of the components to validate the medicine. Random forest algorithm is also helpful for identifying the disease by analyzing the patient's medical records. Random Forest in Medicine As we all know that E-commerce industry is growing at a fast pace. Random Forest algorithm is used for the small segment of recommendation engine for identifying the likely hood of customer liking the recommend products base on the similar kinds of customers.Random Forest for Recommendation System

- **What are the strengths of the model; when does it perform well?** Random Forest can be used for both classification and regression problems. It solves the problem of overfitting and can be used for large datasets. It can be used for feature engineering which means identifying the most important features out of the available features from the training dataset. Random Forests can be easily grown in parallel.

- **What are the weaknesses of the model; when does it perform poorly?** The complexity of Random Forest is one of its disadvantage. They are much harder and time-consuming to construct. The prediction process using random forests is time-consuming than other algorithms.
- **What makes this model a good candidate for the problem, given what you know about the data?** Random Forest is a good candidate for the problem because it require almost no input preparation. They can handle binary features, categorical features, numerical features without any need for scaling. The dataset is quite large therefore it is good to use the algorithm beacuse they are quick to train and work brilliantly when performance optimization happens to enhance model precision.

AdaBoost - Describe one real-world application in industry where the model can be applied. One of the classic use of AdaBoost algorithm is in the face detection. Face recognition is commonly done after a gray scale transformation is done on the RCB image and finally a threshold is assumed to create face boundaries. AdaBoost algorithm was also applied in the detection of Basksetball player. The detailed study of the algorithm can be seen here

- **What are the strengths of the model; when does it perform well?** It is an ensemble method that creates a strong classifier from a number of weak classifiers. AdaBoost was the first really successful boosting algorithm developed for binary classification. It is best used to boost the performance of decision trees on binary classification problems. It is best used with weak learners. It can be less susceptible to the overfitting problem than most learning algorithms.
- **What are the weaknesses of the model; when does it perform poorly?** AdaBoost can be sensitive to noisy data and outliers.
- **What makes this model a good candidate for the problem, given what you know about the data?** This algorithm is a good candidate for the problem because it can boost the performance of the dataset and can give high average accuracy.

Gaussian Naive Bayes (GaussianNB) - Describe one real-world application in industry where the model can be applied. GaussianNB is used in identifying whether an email is spam or not. It is also used to classify a news article about technology, politics, or sports and in checking a piece of text expressing positive emotions, or negative emotions. Practical application of the algorithm can be seen here - **What are the strengths of the model; when does it perform well?** The strengths of the model includes its simplicity and if the NB conditional independence assumption actually holds, a Naive Bayes classifier will converge quicker than discriminative models like logistic regression, so less training data is needed.

- **What are the weaknesses of the model; when does it perform poorly?** The first weakness of the model is that the Naive Bayes classifier makes a very strong assumption on shape of the data distribution and poor performance when independence assumptions doesn't hold.
- **What makes this model a good candidate for the problem, given what you know about the data?** We can learn and predict, given an hypotesis about the "income", which are the most probable donors. This particular model could be a good approach to solving the problem as we have a large data set with few features.

0.6.5 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`. - 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 accuracy_score
from sklearn.metrics import fbeta_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' using
    start = time() # Get start time
    learner.fit(X_train[:sample_size],y_train[:sample_size])
    end = time() # Get end time

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

    # TODO: Get the predictions on the test set(X_test),
    #       then get predictions on the first 300 training samples(X_train) using .pred
    start = time() # Get start time
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_test[:300])
    end = time() # Get end time

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

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

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

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

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

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

        # Return the results
        return results
```

0.6.6 Implementation: Initial Model Evaluation

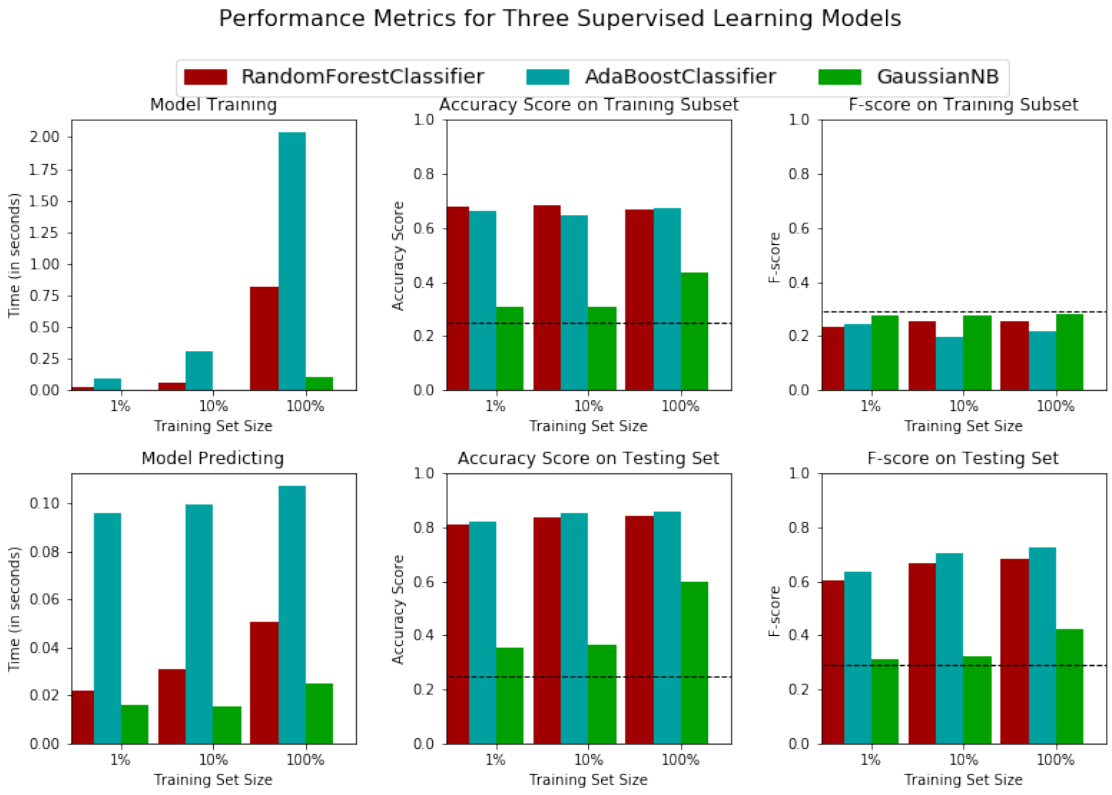
In the code cell, you will need to implement the following: - Import the three supervised learning models you've discussed in the previous section. - Initialize the three models and store them in `'clf_A'`, `'clf_B'`, and `'clf_C'`. - Use a `'random_state'` for each model you use, if provided. - **Note:** 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.ensemble import RandomForestClassifier
         from sklearn.ensemble import AdaBoostClassifier
         from sklearn.svm import SVC
         from sklearn.naive_bayes import GaussianNB


         # TODO: Initialize the three models
         clf_A = RandomForestClassifier(random_state=42)
         clf_B = AdaBoostClassifier(random_state=101)
         clf_C = GaussianNB()

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

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

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

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



```
In [13]: # 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%'}))
```

```
RandomForestClassifier
```

	1%	10%	100%
acc_test	0.808292	0.833610	0.843118
acc_train	0.676667	0.686667	0.666667
f_test	0.603513	0.664743	0.684179
f_train	0.235507	0.257353	0.256410
pred_time	0.022212	0.030941	0.050839
train_time	0.023129	0.058175	0.818944

```
AdaBoostClassifier
```

	1%	10%	100%
acc_test	0.820674	0.849862	0.857601
acc_train	0.663333	0.646667	0.673333
f_test	0.632757	0.701882	0.724551
f_train	0.243506	0.197368	0.220588
pred_time	0.096071	0.099760	0.107335
train_time	0.097002	0.309494	2.034374

```
GaussianNB
```

	1%	10%	100%
acc_test	0.351797	0.366059	0.597678
acc_train	0.306667	0.306667	0.436667
f_test	0.310134	0.320258	0.420899
f_train	0.276268	0.276268	0.284360
pred_time	0.015846	0.015236	0.025220
train_time	0.001864	0.006876	0.099528

0.7 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.

0.7.1 Question 3 - Choosing the Best Model

- Based on the evaluation you performed earlier, in one to two paragraphs, explain to *CharityML* which of the three models you believe to be most appropriate for the task of identifying individuals that make more than $50,000.

** HINT: ** Look at the graph at the bottom left from the cell above(the visualization created by `vs.evaluate(results, accuracy, fscore)`) and check the F score for the testing set when 100% of the training set is used. Which model has the highest score? Your answer should include discussion of the: * metrics - F score on the testing when 100% of the training data is used, * prediction/training time * the algorithm's suitability for the data.

Answer: Choosing the best model - AdaBoost Looking at the graphs above and the tables for three supervised learning models, it is observed that **AdaBoost** performed better than other two models. The F-score for AdaBoost algorithm when 100% of the training set used is 0.724551. The prediction and training time for the algorithm when 100% of the training set used is 0.078021 and 1.617571 respectively. The algorithm is suitable for the data because of its performance and higher accuracy for the given dataset. Since we are more interested in maximizing the accuracy therefore AdaBoost is the best choice as compared to other two models.

0.7.2 Question 4 - Describing the Model in Layman's Terms

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

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

Answer: AdaBoost is a type of "Ensemble Learning" where multiple learners are employed to build a stronger learning algorithm. It is a popular boosting technique which helps to combine multiple "weak classifiers" into a single "strong classifier". A weak classifier is simply a classifier that performs poorly, but performs better than random guessing. We usually use AdaBoost in conjunction with other learning algorithms to improve their performance. Hence the word 'boosting', as in it boosts other algorithms! Boosting is a general method for improving the accuracy of any given learning algorithm. So obviously, adaptive boosting refers to a boosting algorithm that can adjust itself to changing scenarios.

AdaBoost assigns a "weight" to each training example, which determines the probability that each example should appear in the training set. Examples with higher weights are more likely to be included in the training set, and vice versa. After training a classifier, AdaBoost increases the weight on the misclassified examples so that these examples will make up a larger part of the next classifiers training set, and hopefully the next classifier trained will perform better on them.

If each sample consists of a huge number of potential features, the overall system becomes very slow. This is the reason we cannot use a powerful model with a full feature set because it cannot run in real time. We can only afford to have simple machine learning algorithms. But if the algorithms are too simple, they tend to be less accurate. They are called 'weak learners'. So we cascade them together to create a strong classifier. The output of 'weak learners' is combined into a weighted sum that represents the final output of the boosted classifier.

AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. When we collect the set of weak learners, we give

equal weights to all of them. But on each round, the weights of incorrectly classified examples are increased so that the weak learner is forced to focus on the hard examples in the training set.

Consider an example: Let's say we are given the task of listing out all the people in the city of San Francisco who are taller than 5'7, weigh less than 190 lbs, and are between the ages of 28 and 41. Now the problem is that we are supposed to do this without the help of machines. So the only thing we are allowed to do is take a look at the person and determine whether or not that person qualifies. How do we do it? We may or may not be good at estimating these parameters just by looking at the person. So to improve the accuracy, consider three people which will help us in estimating this. The first person is really good at guessing the height, the second person is really good at guessing the weight and the third person is really good at guessing the age. Individually, they may not be all that useful to us, because they can do only one simple task. But if we combine them together and filter out all the people, we have a very good chance of getting an accurate list of people who qualify.This is the concept behind AdaBoost.

AdaBoost is commonly used for face recognition. The article for face recognition using the algorithm can be seen here

0.7.3 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` and `sklearn.metrics.make_scorer`. - 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 [15]: # TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
         from sklearn import grid_search
         from sklearn.metrics import fbeta_score, make_scorer
         from sklearn.grid_search import GridSearchCV

         # TODO: Initialize the classifier
         clf = AdaBoostClassifier(random_state=2)

         # TODO: Create the parameters list you wish to tune, using a dictionary if needed.
         # HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, value2]}
         parameters = {'n_estimators':[75,200,500],'learning_rate':[1.0,1.5,2.0]}

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

         # TODO: Perform grid search on the classifier using 'scorer' as the scoring method usin
```

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

    # 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, prediction
    print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, predictions, beta =
    print("\nOptimized Model\n------")
    print("Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test,
    print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predi
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)


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

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

0.7.4 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.

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

Results: Answer:

- The optimized model's accuracy and F-score on the testing data is 0.8677 and 0.7452 respectively.
- These scores are better than unoptimized model.
- The results obtained from Naive predictor was 0.2478 and 0.2917 for accuracy and F-score respectively. The optimized model produced much better results than the Naive predictor.

0.8 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.

0.8.1 Question 6 - Feature Relevance Observation

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

Answer: I believe that these 5 features should be most important for prediction :

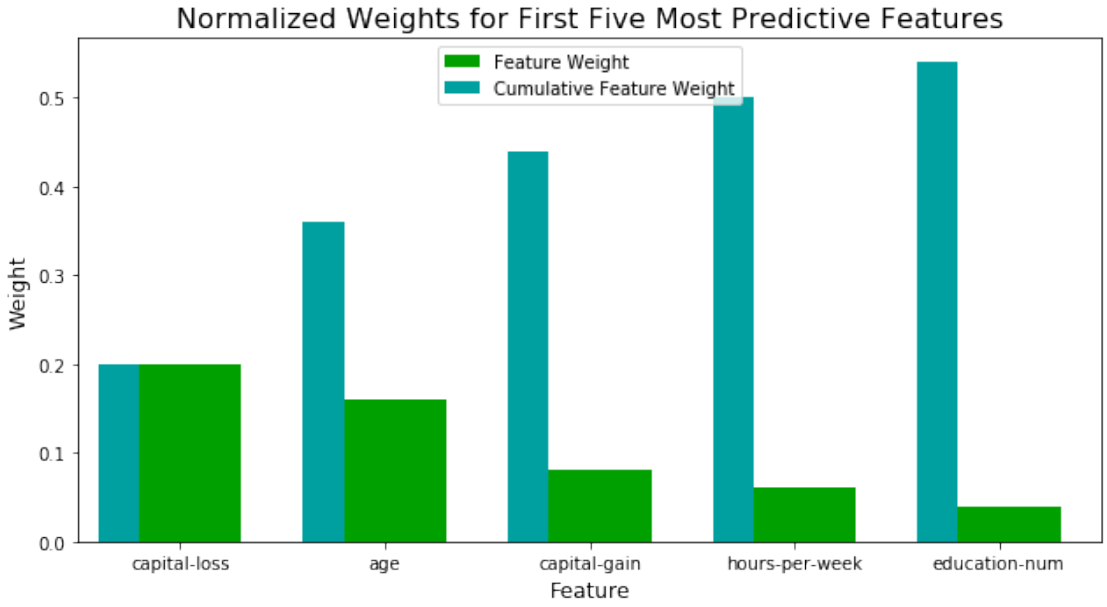
1. **occupation:** Occupation of an individual is one of the important factor because if a person is working as an engineer or doctor his/her income will be high.
2. **capital-gain:** Capital gain measures how much profit an individual is making. Therefore it should be greatly dependent on income.
3. **education:** I believe that people with higher education have better chances of finding higher paying jobs.
4. **age:** Experienced people are paid more in comparision to yoounger generation people.
5. **capital-loss:** Capital loss is also another factor in determining the income of an individual. A capital loss is the loss incurred when a capital asset, such as an investment or real estate, decreases in value. This loss is not realized until the asset is sold for a price that is lower than the original purchase price.

0.8.2 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 [16]:
```
# TODO: Import a supervised learning model that has 'feature_importances_'


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


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

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



0.8.3 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:

- The five features which are most important according to " feature_importances_ " are **education-num, hours-per-week, capital-gain, age and capital-loss**.
- I was partially correct with my predictions. I have not given much importance to hours-per-week feature beacuse I thought occupation should be given higher pereference over it. I have ranked age and capital-loss in 4th and 5th position respectively which remained unaffected. The visualization displays **education-num** as the most important feature which I haven't thought of. I have given it the third rank.
- I have almost predicted the same features but with different ranking. Only one feature which I have not thought of was hours-per-week and given more importance to occupation. I think that hours-per-week is an important factor beacuse if an individual works for more number of hours, he/she will be paid more.

0.8.4 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 [17]:
```
# 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, be
print("\nFinal Model trained on reduced data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduced_predicti
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_predictions,
```

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

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

0.8.5 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 was run on the reduced data (only 5 features out of 13 features) and both the F-score and accuracy score were dropped to 0.7003 and 0.8421 respectively. When the model was trained for full data, both the scores were relatively higher. Therefore, on reducing the number of features, our model's accuracy and F-score were also reduced.

- If training time was a factor, I would still consider the reduced data because AdaBoost is faster as compared to other algorithms and gives accurate results. Training on large sets gives more accurate results than the reduced set but if it is taking more time to process then it is not good. There might be better algorithms that can give good scores for reduced dataset. If we consider the above case, the accuracy and F-score are not affected much as compared to full data. The overall decrease is approximately 0.2 and 0.4 for accuracy and F-score, which can be compromised if training time is taken as a factor.

  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.

##Before You Submit You will also need run the following in order to convert the Jupyter notebook into HTML, so that your submission will include both files.

```
In [ ]: !!jupyter nbconvert *.ipynb
```