Machine Learning Engineer Nanodegree

Supervised Learning

Project: Finding Donors for *CharityML*

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

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

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

Getting Started

In this project, you will employ several supervised algorithms of your choice to accurately model individuals' income using data collected from the 1994 U.S. Census. You will then choose the best candidate algorithm from preliminary results and further optimize this algorithm to best model the data. Your goal with this implementation is to construct a model that accurately predicts whether an individual makes more than $50,000. This sort of task can arise in a non-profit setting, where organizations survive on donations. Understanding an individual's income can help a non-profit better understand how large of a donation to request, or whether or not they should reach out to begin with. While it can be difficult to determine an individual's general income bracket directly from public sources, we can (as we will see) infer this value from other publically available features.

The dataset for this project originates from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Census+Income). The datset was donated by Ron Kohavi and Barry Becker, after being published in the article "*Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid*". You can find the article by Ron Kohavi online (https://www.aaai.org/Papers/KDD/1996/KDD96-033.pdf). The data we investigate here consists of small changes to the original dataset, such as removing the `'fnlwgt'` feature and records with missing or ill-formatted entries.

Exploring the Data

Run the code cell below to load necessary Python libraries and load the census data. Note that the last column from this dataset, `'income'`, will be our target label (whether an individual makes more than, or at most, $50,000 annually). All other columns are features about each individual in the census database.

In [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	occupation	relationship	rac
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	Whi

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 = data.shape[0]

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

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

# TODO: Percentage of individuals whose income is more than $50,000
greater_percent = n_greater_50k/n_records*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_p
ercent))
```

```
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: 24.7843969749237
1%
```

Featureset Exploration

- **age**: continuous.
- **workclass**: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
- **education**: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
- **education-num**: continuous.
- **marital-status**: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
- **occupation**: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
- **relationship**: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
- **race**: Black, White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other.
- **sex**: Female, Male.
- **capital-gain**: continuous.
- **capital-loss**: continuous.
- **hours-per-week**: continuous.
- **native-country**: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

Preparing the Data

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

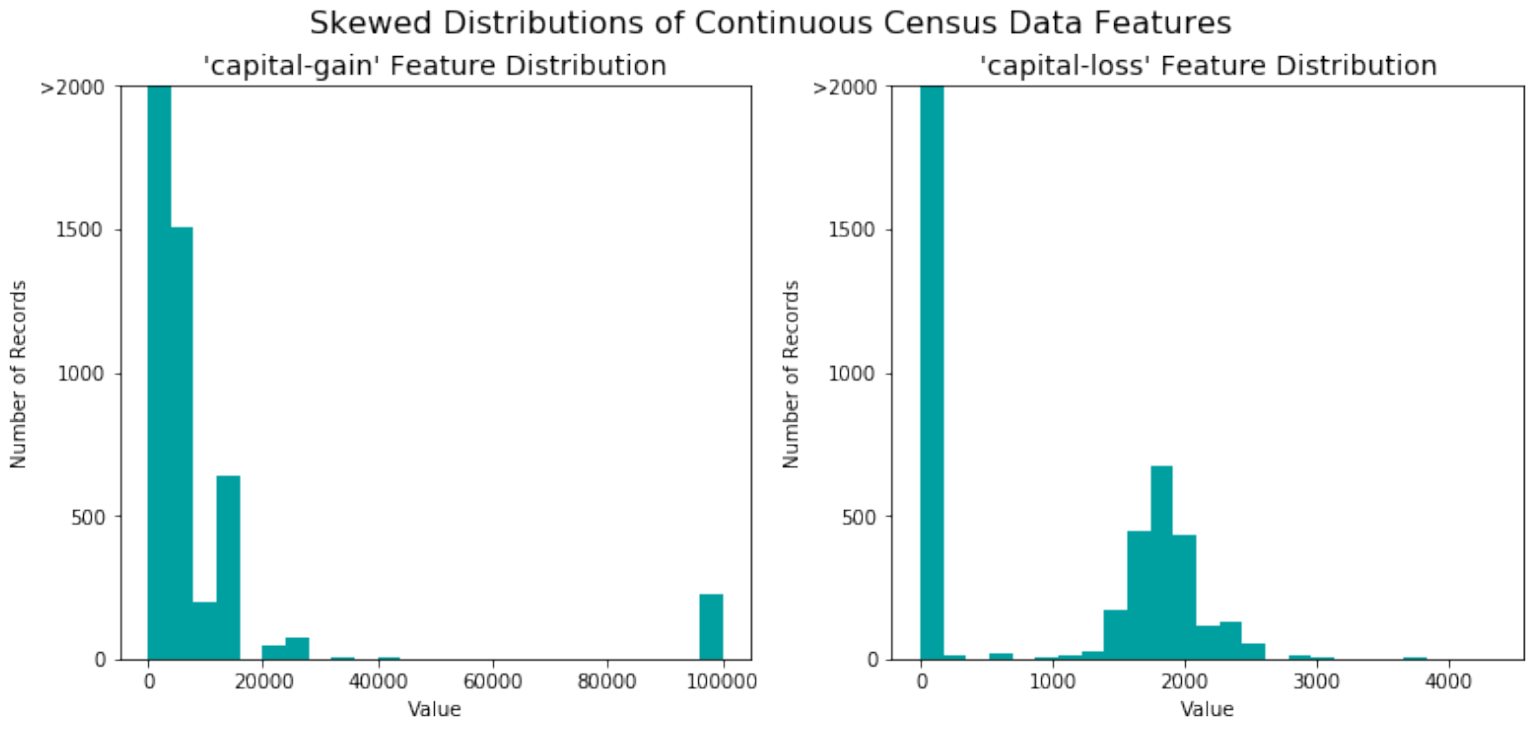
Transforming Skewed Continuous Features

A dataset may sometimes contain at least one feature whose values tend to lie near a single number, but will also have a non-trivial number of vastly larger or smaller values than that single number. Algorithms can be sensitive to such distributions of values and can underperform if the range is not properly normalized. With the census dataset two features fit this description: `'capital-gain'` and `'capital-loss'`.

Run the code cell below to plot a histogram of these two features. Note the range of the values present and how they are distributed.

In [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)
```

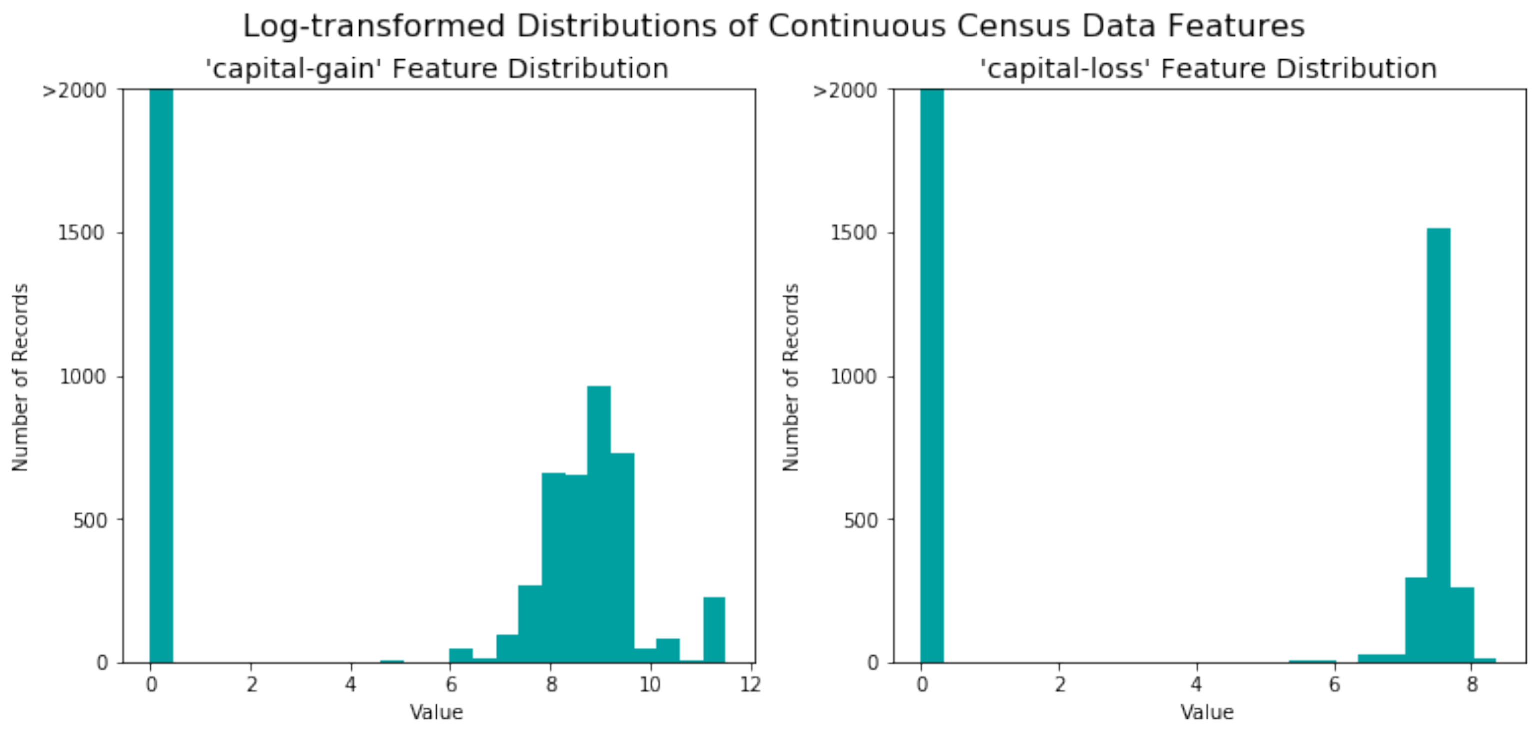


For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'`, it is common practice to apply a logarithmic transformation (https://en.wikipedia.org/wiki/Data_transformation_(statistics)) on the data so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation significantly reduces the range of values caused by outliers. Care must be taken when applying this transformation however: The logarithm of `0` is undefined, so we must translate the values by a small amount above `0` to apply the the logarithm successfully.

Run the code cell below to perform a transformation on the data and visualize the results. Again, note the range of values and how they are distributed.

In [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)
```



Normalizing Numerical Features

In addition to performing transformations on features that are highly skewed, it is often good practice to perform some type of scaling on numerical features. Applying a scaling to the data does not change the shape of each feature's distribution (such as `'capital-gain'` or `'capital-loss'` above); however, normalization ensures that each feature is treated equally when applying supervised learners. Note that once scaling is applied, observing the data in its raw form will no longer have the same original meaning, as exampled below.

Run the code cell below to normalize each numerical feature. We will use [`sklearn.preprocessing.MinMaxScaler` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html)](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html) for this.

In [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_tra
nsformed[numerical])

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

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

Implementation: Data Preprocessing

From the table in **Exploring the Data** above, we can see there are several features for each record that are non-numeric. Typically, learning algorithms expect input to be numeric, which requires that non-numeric features (called *categorical variables*) be converted. One popular way to convert categorical variables is by using the **one-hot encoding** scheme. One-hot encoding creates a "*dummy*" variable for each possible category of each non-numeric feature. For example, assume `someFeature` has three possible entries: `A`, `B`, or `C`. We then encode this feature into `someFeature_A`, `someFeature_B` and `someFeature_C`.

	someFeature		someFeature_A	someFeature_B	someFeature_C
0	B		0	1	0
1	C	----> one-hot encode ---->	0	0	1
2	A		1	0	0

Additionally, as with the non-numeric features, we need to convert the non-numeric target label, `'income'` to numerical values for the learning algorithm to work. Since there are only two possible categories for this label ("<=50K" and ">50K"), we can avoid using one-hot encoding and simply encode these two categories as `0` and `1`, respectively. In code cell below, you will need to implement the following:

- Use `pandas.get_dummies()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html?highlight=get_dummies#pandas.get_dummies) to perform one-hot encoding on the `'features_log_minmax_transform'` data.
- Convert the target label `'income_raw'` to numerical entries.
  - Set records with "<=50K" to `0` and records with ">50K" to `1`.

In [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.map({'>50K':1, '<=50K':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
```

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

Shuffle and Split Data

Now all *categorical variables* have been converted into numerical features, and all numerical features have been normalized. As always, we will now split the data (both features and their labels) into training and test sets. 80% of the data will be used for training and 20% for testing.

Run the code cell below to perform this split.

```
In [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.
```

Evaluating Model Performance

In this section, we will investigate four different algorithms, and determine which is best at modeling the data. Three of these algorithms will be supervised learners of your choice, and the fourth algorithm is known as a *naive predictor*.

Metrics and the Naive Predictor

CharityML, equipped with their research, knows individuals that make more than $50,000 are most likely to donate to their charity. Because of this, *CharityML* is particularly interested in predicting who makes more than $50,000 accurately. It would seem that using **accuracy** as a metric for evaluating a particular model's performace would be appropriate. Additionally, identifying someone that *does not* make more than $50,000 as someone who does would be detrimental to *CharityML*, since they are looking to find individuals willing to donate. Therefore, a model's ability to precisely predict those that make more than $50,000 is *more important* than the model's ability to **recall** those individuals. We can use **F-beta score** as a metric that considers both precision and recall:

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

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

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

Note: Recap of accuracy, precision, recall

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

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

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

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

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

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

Question 1 - Naive Predictor Performace

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

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

HINT:

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

In [12]:

```
'''
TP = np.sum(income) # Counting the ones as this is the naive case. Note that 'in
come' is the 'income_raw' data
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 = TP/(TP+FP)
recall = TP/(TP+FN)
precision = TP/(TP+FP)

# TODO: Calculate F-score using the formula above for beta = 0.5 and correct val
ues 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(accura
cy, fscore))
```

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

Supervised Learning Models

The following are some of the supervised learning models that are currently available in `scikit-learn` (http://scikit-learn.org/stable/supervised_learning.html) that you may choose from:

- Gaussian Naive Bayes (GaussianNB)
- Decision Trees
- Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting)
- K-Nearest Neighbors (KNeighbors)
- Stochastic Gradient Descent Classifier (SGDC)
- Support Vector Machines (SVM)
- Logistic Regression

Question 2 - Model Application

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

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

HINT:

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

Answer: I choose Gaussian Naive Bayes (GaussianNB), Support Vector Machines (SVM), and AdaBoost.

GaussianNB:

1. Real-world application:
    - Document classification, such as email spam filtering.
2. Strengths:
    - Very good at binary classification.
    - Can be much faster/cheaper to run than more complex algorithms.
    - Doesn't require a lot of training data.
    - Able to take in a large number of features without a drop-off in score.
    - Easy to implement; usually not necessary to tune hyperparameters.
3. Weaknesses:
    - Bad at clasifying texts where phrases (such as "Chicago Bulls") have meanings that are very different from the combined individual meanings of their component words (bulls, animals, that are in or around the city of Chicago). This is because Naive Bayes assumes that each feature (each word, in this case) is independent.
    - A poor estimator (the result of the predict_proba method usually shouldn't be trusted).
4. Why a good candidate:
    - Naive Bayes will be a great baseline algorithm to serve as a benchmark for the performance of my other two algorithms. Even with 103 features in my data after one-hot encoding, I should be able to implement and execute Naive Bayes with relative ease, as I don't have to worry about feature selection or tuning hyperparameters. Indeed, because Naive Bayes is very good at binary classification right out of the box, I expect to get fairly high score with a minimum of effort. This baseline should push me to get the most out of my other two algorithms.
5. References:
    - https://en.wikipedia.org/wiki/Naive_Bayes_classifier (https://en.wikipedia.org/wiki/Naive_Bayes_classifier)
    - http://scikit-learn.org/stable/modules/naive_bayes.html (http://scikit-learn.org/stable/modules/naive_bayes.html)

- http://www.cs.unb.ca/~hzhang/publications/FLAIRS04ZhangH.pdf (http://www.cs.unb.ca/~hzhang/publications/FLAIRS04ZhangH.pdf)
- https://youtu.be/nfbKTrufPOs (https://youtu.be/nfbKTrufPOs)

SVM:

1. Real-world application:
   - Image classification
   - Hand-written character recognization
2. Strengths:
   - Good for both classification and regression analysis.
   - Effective in high dimensional spaces, even when the number of dimensions is greater than the number of samples.
   - Works best in domains where there is a wide margin of separation in the data.
   - Very versatile. Can use one of the common kernels (rbf, polynomial, linear) or even build a custom kernel.
3. Weaknesses:
   - If dataset is massive, will be more slow/expensive to execute.
   - Can be prone to overfitting if data has a lot of noise.
   - More time needs to be spent to tune hyperparameters.
   - Doesn't directly provide probability estimates (if needed, they must be calculated using the more costly five-fold cross-validation.
4. Why a good candidate:
   - I chose SVM becuase of its reputation for both being very good at classification, and being effective in high dimensions, given that my dataset has 103 features. I am a little concerned about how fast the algorithm will run, seeing as how my dataset has over 45,000 entries, but I am willing to take this risk in order to be able to take advantage of SVM's tuning/customization capabilities (such as the gamma parameter of the rbf kernel) in the hope that I can beat my Naive Bayes classifier's score.
5. References:
   - https://en.wikipedia.org/wiki/Support_vector_machine (https://en.wikipedia.org/wiki/Support_vector_machine)
   - http://scikit-learn.org/stable/modules/svm.html (http://scikit-learn.org/stable/modules/svm.html)
   - https://youtu.be/U9-ZsbaaGAs (https://youtu.be/U9-ZsbaaGAs)

AdaBoost:

1. Real-world application:
   - Face detection (which is basically just a more complex form of boundary detection)
2. Strengths:
   - Can be used in both classification and regression problems.
   - During training, selects only the features that improve the model's predictive power, which reduces dimensionality, and could help speed up the algorithm's execution time.
   - When used with a decision tree as the weak learner, AdaBoost requires very little extra customization (unlike SVM).
   - Often less susceptible to overfitting than other algorithms (thanks to AdaBoost's ability to

assign higher weights to the best performing weak learners).

3. Weaknesses:
   - Can sometimes still be sensitive to very noisy data and outliers.
4. Why a good candidate:
   - I chose AdaBoost because it is good for classification, and because it is an adaptive learner -- as it puts higher weights on the best performing weak learners (decision trees, in this case), unimportant features automatically get left by the wayside. This saves me from having to consciously decide which of my dataset's 103 features will be most relevant. Finally, AdaBoost with decision trees as weak learners works pretty well right out of the box, which should save me from having to spend a lot of time tuning it.
5. References:
   - http://scikit-learn.org/stable/modules/ensemble.html#adaboost (http://scikit-learn.org/stable/modules/ensemble.html#adaboost)
   - https://www.cs.princeton.edu/~schapire/papers/explaining-adaboost.pdf (https://www.cs.princeton.edu/~schapire/papers/explaining-adaboost.pdf)
   - https://en.wikipedia.org/wiki/AdaBoost (https://en.wikipedia.org/wiki/AdaBoost)
   - https://www.analyticsvidhya.com/blog/2015/05/boosting-algorithms-simplified/ (https://www.analyticsvidhya.com/blog/2015/05/boosting-algorithms-simplified/)

Implementation - Creating a Training and Predicting Pipeline

To properly evaluate the performance of each model you've chosen, it's important that you create a training and predicting pipeline that allows you to quickly and effectively train models using various sizes of training data and perform predictions on the testing data. Your implementation here will be used in the following section. In the code block below, you will need to implement the following:

- Import `fbeta_score` and `accuracy_score` from `sklearn.metrics` (http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics).
- Fit the learner to the sampled training data and record the training time.
- Perform predictions on the test data `X_test`, and also on the first 300 training points `X_train[:300]`.
  - Record the total prediction time.
- Calculate the accuracy score for both the training subset and testing set.
- Calculate the F-score for both the training subset and testing set.
  - Make sure that you set the `beta` parameter!

In [13]:

```
# TODO: Import two metrics from sklearn - fbeta_score and accuracy_score
from sklearn.metrics import fbeta_score, accuracy_score

def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
    '''
    inputs:
       - learner: the learning algorithm to be trained and predicted on
       - sample_size: the size of samples (number) to be drawn from training 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 .fit(training_features[:], training_labels[:])
    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) usin
g .predict()
    start = time() # Get start time
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train[:300])
    end = time() # Get end time

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

    # TODO: Compute accuracy on the first 300 training samples which is y_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_sc
ore()
    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
```

Implementation: Initial Model Evaluation

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

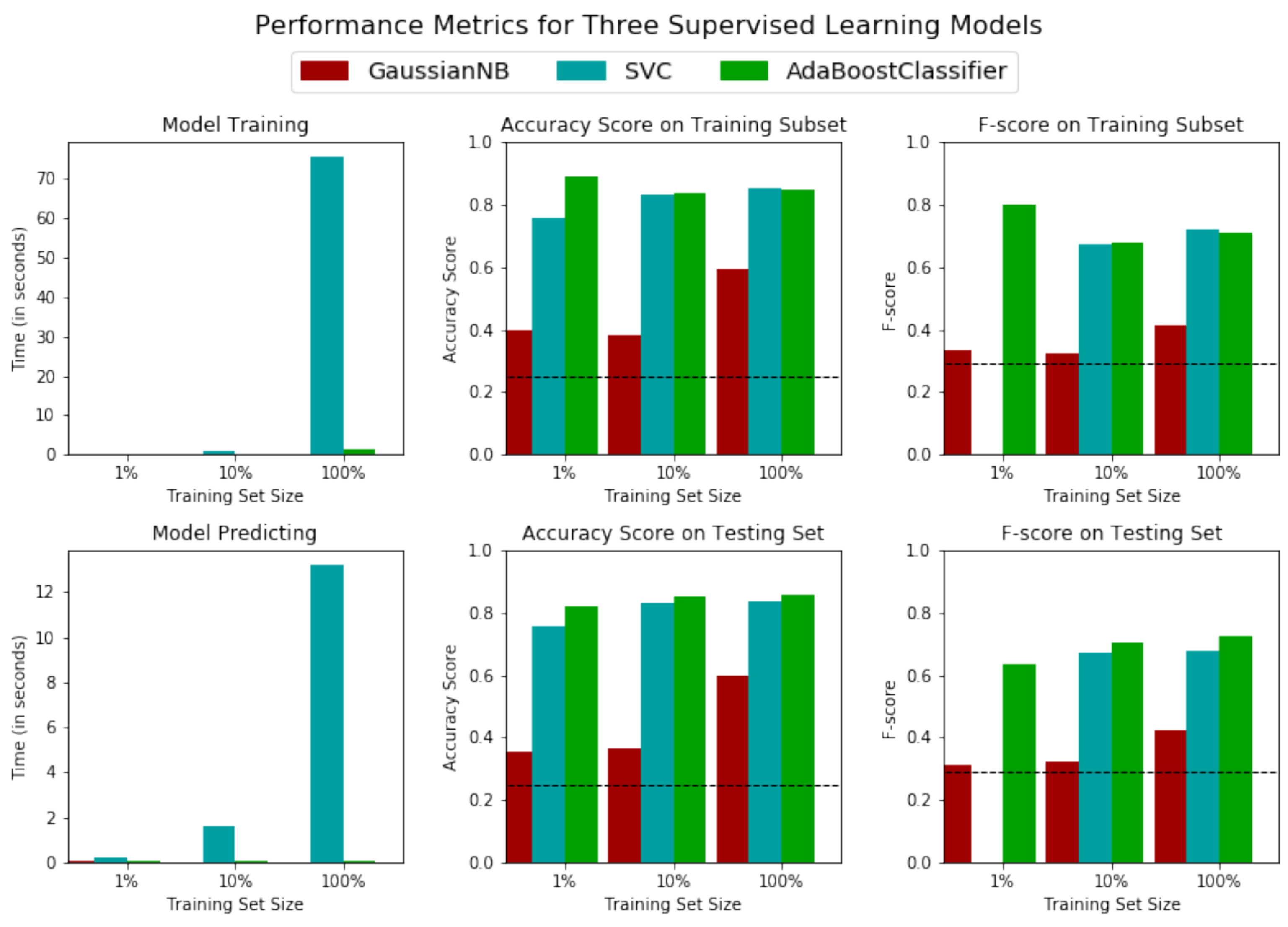
- Import the three supervised learning models you've discussed in the previous section.
- Initialize the three models and store them in `'clf_A'`, `'clf_B'`, and `'clf_C'`.
  - Use a `'random_state'` for each model you use, if provided.
  - **Note:** Use the default settings for each model — you will 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 [14]:

```
# TODO: Import the three supervised learning models from sklearn
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier

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

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the training da
ta
# 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 `int` and not `float`)
# HINT: samples_1 is 1% of samples_100 (ensure to set the count of the values to
be `int` and not `float`)
samples_100 = len(y_train)
samples_10 = int(.1*samples_100)
samples_1 = int(.01*samples_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)
```

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

/anaconda3/envs/finding_donors_project/lib/python3.5/site-packages/s
klearn/metrics/classification.py:1074: UndefinedMetricWarning: F-sco
re is ill-defined and being set to 0.0 due to no predicted samples.
  'precision', 'predicted', average, warn_for)

SVC trained on 361 samples.
SVC trained on 3617 samples.
SVC trained on 36177 samples.
AdaBoostClassifier trained on 361 samples.
AdaBoostClassifier trained on 3617 samples.
AdaBoostClassifier trained on 36177 samples.
```



Improving Results

In this final section, you will choose from the three supervised learning models the *best* model to use on the student data. You will then perform a grid search optimization for the model over the entire training set (`X_train` and `y_train`) by tuning at least one parameter to improve upon the untuned model's F-score.

Question 3 - Choosing the Best Model

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

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

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

Answer: AdaBoost is the most appropriate model for the task of identifying individuals that make more than $50,000. The most important metric for evaluating our models is the testing data F-score when 100% of the training data is used. Using 100% of the training data gives us the best possible chance of observing convergence between the testing and training score curves and avoiding a classifier that exhibits high bias (underfitting) or high variance (overfitting). Using an F-score metric with a beta value of 0.5 allows us to score our model based on its precision and recall, with heaviest emphasis placed on the model's precision. This is appropriate because the purpose of our model is to guess whether or not an individual makes more than $50,000, and precision is the the probability that our model is correct when it guesses that someone earns more than $50,000 (recall indicates the fraction of true +$50,000 earners that our model is able to find).

Looking at the graph above, we see that our AdaBoost and SVM models had the highest testing data F-scores when 100% of the training data was used. The Naive Bayes model performed only slightly better than our naive predictor (if we were to simply guess all individuals in the dataset made more than $50,000). In fact, when less than 100% of the training data was used, the Naive Bayes model actually performed worse than our naive predictor. We can thus safely conclude that the Naive Bayes model won't be a good predictor.

AdaBoost had the highest F-score, and our SVM model wasn't too far behind. All else being equal, I would expect that with enough hyperparameter tuning, it might be possible to get the SVM model to perform better than AdaBoost. However, it takes much longer to train and predict with SVM than with AdaBoost, and I don't believe that any extra performance we might be able to squeeze out of our SVM model would be worth the tradeoff in terms of extra time and expense that would be necessary in order to run SVM instead of AdaBoost.

Moreover, AdaBoost does a good job of not overfitting to the noise in our dataset. The fact that the the AdaBoost model's testing data F-score is about the same as its training data subset F-score is evidence of this. SVM has a harder time avoiding overfitting to our dataset's noise.

All in all, AdaBoost gives us the biggest bang for the buck. It has the highest F-score, and so we see that it does the best job of quickly navigating through all of our dataset's 103 features and determining the most important signals (weak learners) to pay attention to when guessing whether an individual earns more than $50,000.

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: Our implementation of AdaBoost works by fitting several Decision Tree Classifiers to our dataset in succession. More generally, these classifiers are called "weak learners." Each weak learner's mission is to learn just a little bit about our data -- enough just to be able to make a quick guess about how to split the data into two separate categories (in our case, people who earn more than $50,000 and those who earn $50,000 or less).

Now each time a weak learner is fit to our data, AdaBoost measures its accuracy and adds an entry to a list that it's keeping of all the weak learners and their respective accuracies. Furthermore, in between rounds (after one weak learner completes its work and before the next weak learner starts), AdaBoost places higher weights on all the points in our data that the previous weak learner failed to properly categorize. This in turn forces the next weak learner to put most of its effort into properly categorizing this set of previously miscategorized points.

We can imagine that AdaBoost has a big spotlight that it can shine on any part of the dataset. Each weak learner will pay the most attention to properly categorizing the parts of the dataset that the spotlight is illuminating. After each round AdaBoost decides what part of the dataset has been most neglected by the previous weak learners and shines its spotlight on that part.

We can have as many of these weak learners as we want. So far, we've been using 50 of them, which is the default. Once they've all been run, AdaBoost combines the small amounts of information that each weak learner has learned about our data in order to draw a complex boundary that will be used to categorize our data into its two categories (earning more than $50K, or earning $50K or less). This complex boundary is a composite of all the simple boundaries, or predictions, that were drawn by each of our weak learners.

The final issue is that some of these weak learners had made smarter predictions and some hadn't done so well. So which weak learners should AdaBoost listen to, and in what order? AdaBoost figures this out by looking back at the list that it's been keeping of each weak learner's accuracy. (Accuracy is the fraction of points in the dataset that a weak learner properly categorized.) AdaBoost places higher weights on the predictions made by weak learners that had higher accuracy scores. And when combining all the weak learners' individual predictions into one composite final prediction, those higher weights ensure that AdaBoost will pay the most attention to weak learners that had the highest accuracy scores, and less attention to those that didn't.

Reference: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html (http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html)

Implementation: Model Tuning

Fine tune the chosen model. Use grid search (`GridSearchCV`) with at least one important parameter tuned with at least 3 different values. You will need to use the entire training set for this. In the code cell below, you will need to implement the following:

- Import `sklearn.grid_search.GridSearchCV` (http://scikit-learn.org/0.17/modules/generated/sklearn.grid_search.GridSearchCV.html) and `sklearn.metrics.make_scorer` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- Initialize the classifier you've chosen and store it in `clf`.
  - Set a `random_state` if one is available to the same state you set before.
- Create a dictionary of parameters you wish to tune for the chosen model.
  - Example: `parameters = {'parameter' : [list of values]}`.
  - **Note:** Avoid tuning the `max_features` parameter of your learner if that parameter is available!
- Use `make_scorer` to create an `fbeta_score` scoring object (with $\beta = 0.5$).
- Perform grid search on the classifier `clf` using the `'scorer'`, and store it in `grid_obj`.
- Fit the grid search object to the training data (`X_train`, `y_train`), and store it in `grid_fit`.

Note: Depending on the algorithm chosen and the parameter list, the following implementation may take some time to run!

In [24]:

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

# TODO: Initialize the classifier
clf = AdaBoostClassifier(random_state=42)

# TODO: Create the parameters list you wish to tune, using a dictionary if neede
d.
# HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, v
alue2]}
parameters = {'n_estimators': [1000,1100,1200,1300]}

# 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 meth
od using GridSearchCV()
grid_obj = GridSearchCV(clf, parameters, scoring=scorer)

# TODO: Fit the grid search object to the training data and find the optimal par
ameters using fit()
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, pre
dictions)))
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, bes
t_predictions, beta = 0.5)))
print("\nClassifier for Optimized Model\n------")
print(best_clf)
```

```
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: 0.8679
Final F-score on the testing data: 0.7460

Classifier for Optimized Model
------
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=1200, random_state=42)
```

Question 5 - Final Model Evaluation

- What is your optimized model's accuracy and F-score on the testing data?
- Are these scores better or worse than the unoptimized model?
- How do the results from your optimized model compare to the naive predictor benchmarks you found earlier in **Question 1**?

Note: Fill in the table below with your results, and then provide discussion in the **Answer** box.

Results:

Metric	Unoptimized Model	Optimized Model
Accuracy Score	0.8576	0.8679
F-score	0.7246	0.7460

Answer:

- My optimized model's accuracy score on the testing data is 0.8679; its F-score on the testing data is 0.7460.
- My optimized model's accuracy score is about 1 percentage point higher than that of the unoptimized model; its F-score is about 2 percentage points higher than that of the unoptimized model.
- The naive predictor from **Question 1** had an accuracy score of 0.2478 and an F-score of 0.2917. My optimized model does much better, with an accuracy score that is about 3.5 times as large as the naive predictor's accuracy score, and an F-score that is two and a half times as large as that of the naive predictor.

Feature Importance

An important task when performing supervised learning on a dataset like the census data we study here is determining which features provide the most predictive power. By focusing on the relationship between only a few crucial features and the target label we simplify our understanding of the phenomenon, which is most always a useful thing to do. In the case of this project, that means we wish to identify a small number of features that most strongly predict whether an individual makes at most or more than $50,000.

Choose a scikit-learn classifier (e.g., adaboost, random forests) that has a `feature_importance_` attribute, which is a function that ranks the importance of features according to the chosen classifier. In the next python cell fit this classifier to training set and use this attribute to determine the top 5 most important features for the census dataset.

Question 6 - Feature Relevance Observation

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

Answer:

My top 5 most relevant features for predicting a person's income (in order of decreasing importance):

1. **Occupation**: It's a simple fact that income levels track very closely with job type. It's essentially a universal truth that someone with an occupation categorized as 'Exec-managerial' would make more money than an individual who falls into the 'Handlers-cleaners' category. Occupation is therefore the first feature I would look to when predicting an individual's income level.
2. **Education Level**: After occupation, I believe that education level is the next best predictor. Higher paying professional and technical jobs typically require longer periods of training, the product of which is a higher level of education. I could have chosen to look at the number of years of education instead of education level, and expect I would see similar results. However, when push comes to shove, I believe education level will be a bit more informative. People typically must have completed their degrees in order to obtain higher paying jobs.
3. **Capital Gain**: Folks with higher incomes often are able to set aside more of their earnings to invest in things like real estate or the stock market. I would thus expect higher income earners to have a greater level of capital gains from their investments. Looking at capital loss is an alternative way to observe this correlation. After all, simply being able to invest lots of money doesn't mean one will automatially earn greater returns. Those who invest more could conceivably lose more, as well. However, again when push comes to shove, I'm betting that capital gain will be the better feature to look at -- my intuition is that folks who earn more, and thus invest more, have sounder/savvier investing strategies than those who earn less and thus invest less, and are therefore a bit more likely to experience capital gain than capital loss.
4. **Relationship**: Certain relationship patterns, such as having a wife or a husband or a child, are said to correlate with a more stable, postive home life, which itself could correlate with an individual earning more. This pattern alone might be enough to justify using the relationship feature. Yet there's one more reason why relationship may be a good predictor of earnings: individuals who have certain familial relationships, like children, for example, likely earn more than those who don't. I would be willing to guess that people who are responsible for supporting other family members earn more on the whole than those who aren't.
5. **Age**: In most fields, the more experience you have, the more money you earn. More experience requires more years of work, which usually means that those who have this extended experience are older than those who lack it. However, one trend that works against this feature is that there could be individuals, such as long time homemakers or caregivers, who are older but have never worked or have worked very little. While I think the pros of using this feature ultimately outweigh the cons, this particular drawback leads me to rank age lowest out of my top 5 features.

Implementation - Extracting Feature Importance

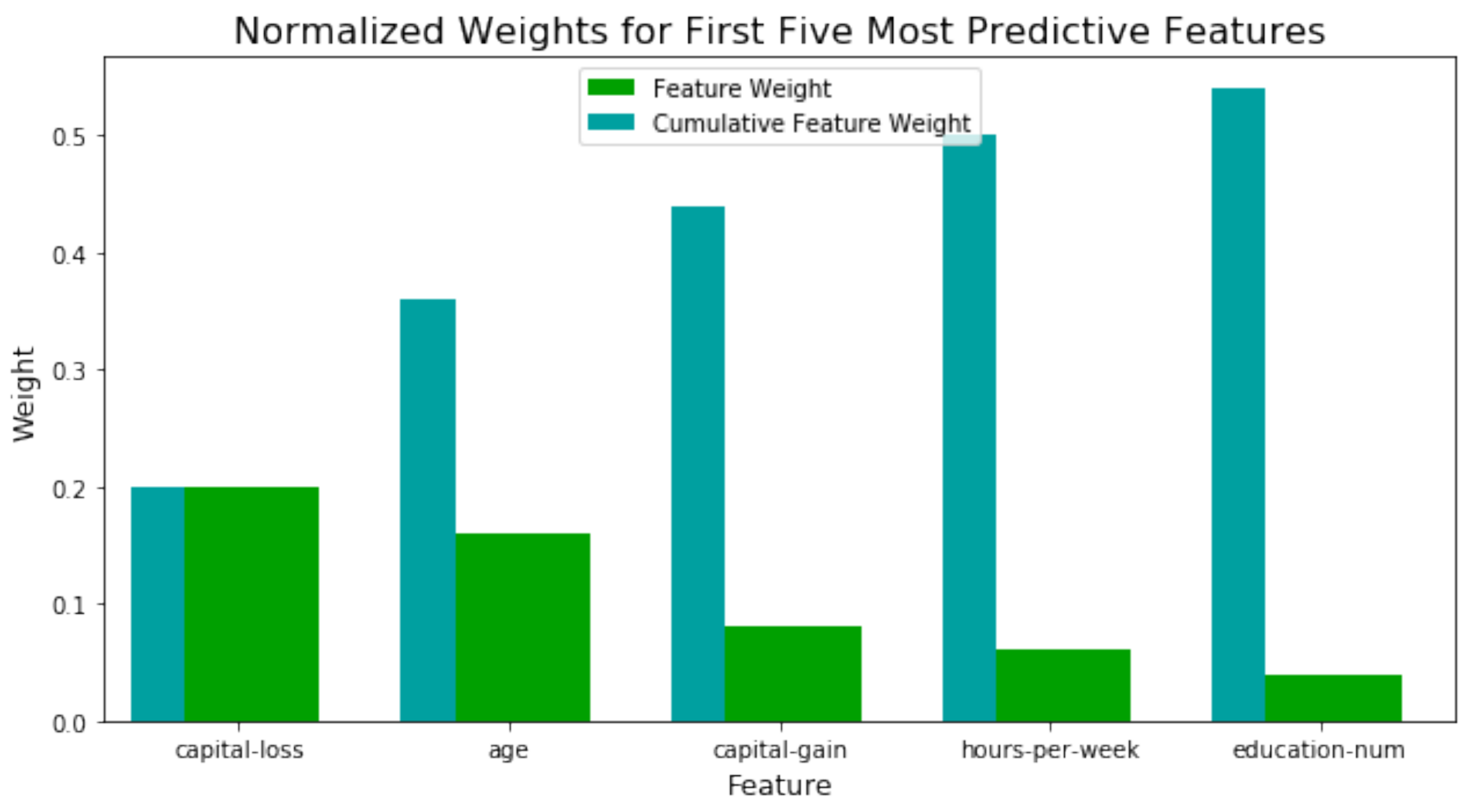
Choose a `scikit-learn` supervised learning algorithm that has a `feature_importance_` attribute availble for it. This attribute is a function that ranks the importance of each feature when making predictions based on the chosen algorithm.

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 [26]:

```
# TODO: Import a supervised learning model that has 'feature_importances_'
from sklearn.ensemble import AdaBoostClassifier

# TODO: Train the supervised model on the training set using .fit(X_train, y_tra
in)
model = AdaBoostClassifier(random_state=42)
model.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)
```



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:

- I was not surprised to see that age and capital-gain made it into the top 5 feature importances of the AdaBoost classifier I chose above, as these were two of the five features I had already expected would be important. The only difference is that it appears that age is more important than I had expected, and my concern about there being too many folks who were older but had minimal or no work histories and thus lower wages (thus counterbalancing those who had more work experience and higher wages) was unfounded.
- Furthermore, since I had already chosen education_level, I wasn't surprised to see that education-num also made it into the top 5 here. The only difference between my intuition regarding education and AdaBoost's feature importances is that absolute number of years of education apparently does a better job of predicting income than completed education level does.
- Although I did previously choose to go with capital-gain, I acknowledged that capital-loss could be almost as reasonable a predictor of income. Apparently, according to AdaBoost's feature importances, capital-loss is not only more important than capital-gain, it is the most important feature for predicting income. This tells me that higher earners either have substantially more capital losses than do low earners, or else that the reverse is true. I would bet that the former is the case, as more money invested ultimately results in either higher earnings or higher losses. My mistake was not appreciating the importance of my intuition regarding both capital-gains and capital-losses and including both features in my top 5.
- All in all, I could make the case that I was expecting four out of the five features that AdaBoost ultimately chose. My biggest surprise: occupation and relationship didn't make it into the top 5, and hours-per-week did. I initially neglected hours-per-week because my hunch was that it would correlate with folks who were in lower working classes, and therefore lower income earners. I believe that the mistake I made was to conflate hours-per-week with "hourly workers." I forgot that while they are not paid by the hour, professional or salaried workers are still aware of and able to report the number of hours they work, and that this number of hours may well be positively correlated with the salaries of these workers' more demanding, higher paying jobs. In retrospect, it now makes sense to me that hours-per-week is more important than occupation.
- My intuition about relationship being an important feature was clearly wrong. Again, I wish I had doubled-down on my intuition regarding capital-gains/losses and included both of those features in my original top 5.

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 [27]:
```

```
# 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_pred
ictions)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, best_predicti
ons, beta = 0.5)))
print("\nFinal Model trained on reduced data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduced_p
redictions)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_predi
ctions, beta = 0.5)))
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8679
F-score on testing data: 0.7460

Final Model trained on reduced data
------
Accuracy on testing data: 0.8429
F-score on testing data: 0.7034
```

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:

- When trained on only five features, the final model's F-score and accuracy score are both lower than when all features are used: when trained on the full data, accuracy is 0.8679 and F-score is 0.7460; when trained on only 5 features, accuracy is 0.8429 and F-score is 0.7034.

- Seeing as how I used an AdaBoost classifier, I'm not surprised that this turned out to be the case, one of AdaBoost's advantages is having less of a tendency to overfit to training data, especially when many features are used. If I had used an SVM model instead, I can easily imagine that training the model on only five features may have resulted in a higher performing classifier. This is because SVM is much more sensitive to noise in a dataset.

- Whether or not I would consider using the reduced data as my training set depends on just how much of a factor training time is. In this particular case, training time is really not a factor as the AdaBoost algorithm is able to fit and predict in a much shorter amount of time than another algorithm like SVM. My best-performing AdaBoost classifier uses 1,200 estimators, and this takes just under a minute to run on my computer when training on all the dataset's features. Training on only five features is nearly instantaneous. I am more than willing to spend this small amount of extra time (just under one minute extra) in order to get an F-score that is more than 4 percentage points higher than that which I get from fitting my AdaBoost classifier to just five features.

- Nonetheless, hypothetically speaking, if training time were a huge factor, I would be more than willing use the reduced data as my training set. The penalty of a 4 percentage point drop in F-score would be a price I'd be willing to pay in order to avoid spending an extra several minutes or possibly hours training my model.

> **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.