finding_donors

January 4, 2018

1 Machine Learning Engineer Nanodegree

1.1 Supervised Learning

1.2 Project: Finding Donors for *CharityML*

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

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

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

1.3 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.

1.4 Exploring the Data

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

```
In [1]: # Import libraries necessary for this project
        import numpy as np
        import pandas as pd
        from time import time
        from IPython.display import display # Allows the use of display() for DataFrames

        # Import supplementary visualization code visuals.py
        import visuals as vs

        # Pretty display for notebooks
        %matplotlib inline

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

        # Success - Display the first record
        display(data.head(n=1))
```

```
   age   workclass education_level  education-num  marital-status  \
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
```

1.4.1 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 [2]: # TODO: Total number of records
        n_records = len(data[[0]])

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

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

        # TODO: Percentage of individuals whose income is more than $50,000
        greater_percent = float(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: {:.2f}%".format(greater_per
```

```
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%
```

1.5 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.

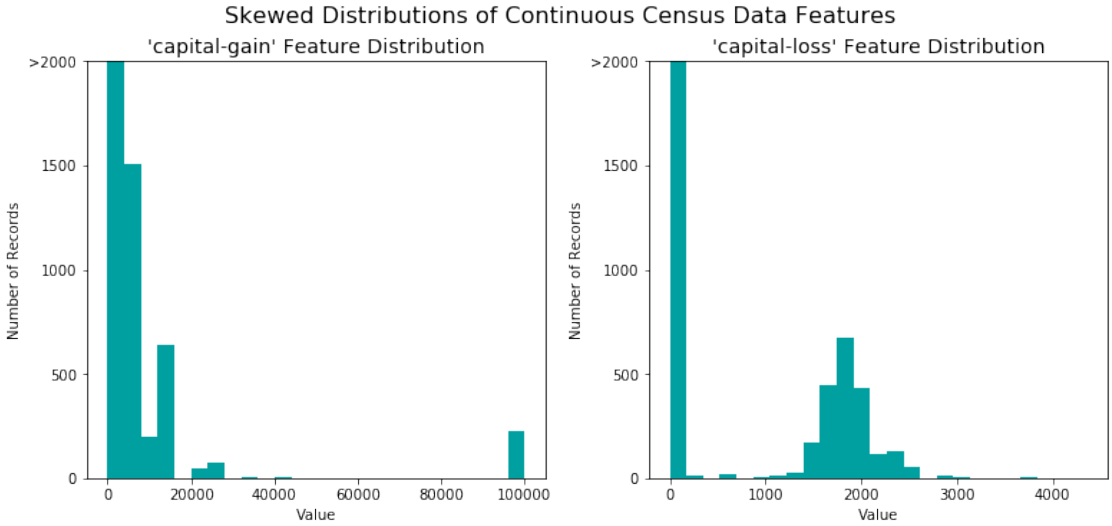
1.5.1 Transforming Skewed Continuous Features

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 [3]: # Split the data into features and target label
        income_raw = data['income']
```

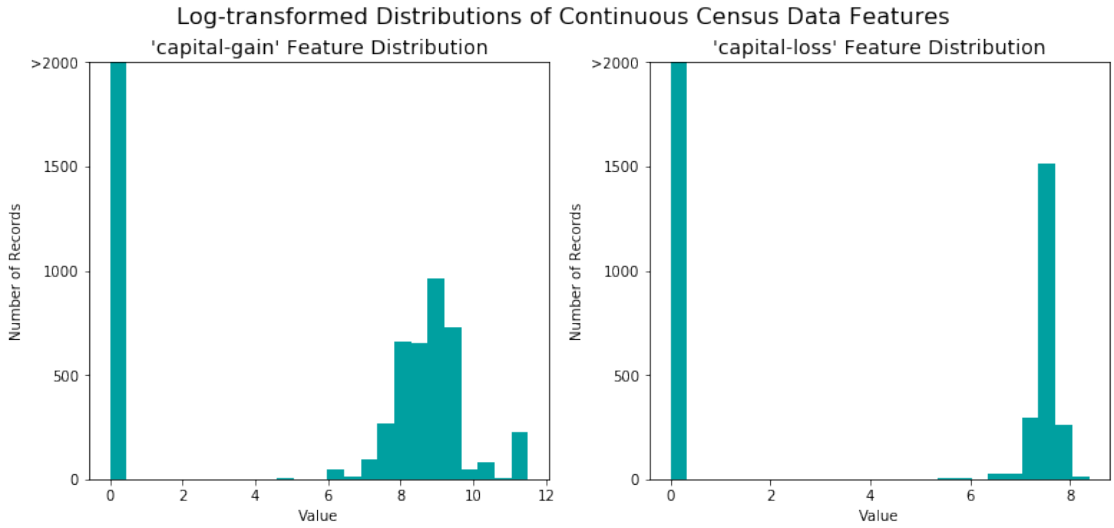
```
features_raw = data.drop('income', axis = 1)

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



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

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

In [4]:
```
# Log-transform the skewed features
skewed = ['capital-gain', 'capital-loss']
features_raw[skewed] = data[skewed].apply(lambda x: np.log(x + 1))

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



1.5.2 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 [5]: # Import sklearn.preprocessing.StandardScaler
        from sklearn.preprocessing import MinMaxScaler

        # Initialize a scaler, then apply it to the features
        scaler = MinMaxScaler()
        numerical = ['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week']
        features_raw[numerical] = scaler.fit_transform(data[numerical])

        # Show an example of a record with scaling applied
        display(features_raw.head(n = 1))
```

```
       age   workclass education_level  education-num  marital-status  \
0  0.30137   State-gov       Bachelors            0.8   Never-married

      occupation    relationship    race    sex  capital-gain  capital-loss  \
0   Adm-clerical   Not-in-family   White   Male       0.02174           0.0

   hours-per-week  native-country
0        0.397959   United-States
```

1.5.3 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_raw'` data. - Convert the target label `'income_raw'` to numerical entries. - Set records with "<=50K" to `0` and records with ">50K" to `1`.

```
In [6]: # TODO: One-hot encode the 'features_raw' data using pandas.get_dummies()
        features = pd.get_dummies(features_raw)

        # TODO: Encode the 'income_raw' data to numerical values
        income = pd.get_dummies(income_raw)[[1]]

        # Print the number of features after one-hot encoding
        encoded = list(features.columns)
        print "{} total features after one-hot encoding.".format(len(encoded))

        # Uncomment the following line to see the encoded feature names
        print encoded
```

```
103 total features after one-hot encoding.
['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week', 'workclass_ Federal-
```

1.5.4 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 [7]: # Import train_test_split
        from sklearn.cross_validation import train_test_split

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

        # 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.


C:\Users\Tron\Anaconda2\lib\site-packages\sklearn\cross_validation.py:44: DeprecationWarning: T
  "This module will be removed in 0.20.", DeprecationWarning)
```

1.6 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*.

1.6.1 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.

1.6.2 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?
Note: You must use the code cell below and assign your results to `'accuracy'` and `'fscore'` to be used later.

In [8]:
```
# TODO: Calculate accuracy
accuracy = float(n_greater_50k)/n_records

# TODO: Calculate F-score using the formula above for beta = 0.5
precision = float(n_greater_50k)/n_records
recall = float(n_greater_50k)/n_greater_50k
beta = 0.5
fscore = (1 + beta**2)*precision*recall/(beta**2*precision + recall)

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

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

1.6.3 Supervised Learning Models

The following supervised learning models 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

1.6.4 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.* (You may need to do research for this — give references!) - *What are the strengths of the model; when does it perform well? - What are the weaknesses of the model; when does it perform poorly? - What makes this model a good candidate for the problem, given what you know about the data?*

Answer:
I SVM

```
I.1 Applications:
    SVM outperforms backpropagation neural network for financial time series forecasting
    TAY, Francis E. H. and Lijuan CAO, 2001. Application of support vector machines in financia

    CHEN, Wun-Hwa and Jen-Ying SHIH, 2006. A study of Taiwan's issuer credit rating systems usi

    CHEN, Wun-Hua, Jen-Ying SHIH and Soushan WU, 2006. Comparison of support-vector machines ar
```

Industrial application for mechanical faults diagnostic
Lane Maria Rabelo Baccarini, , Valceres Vieira Rocha e Silva, Benjamim Rodrigues de Meneze

I.2 Advantages:
Robust to noise
http://condor.depaul.edu/ntomuro/courses/578/notes/SVM-overview.pdf (SVM overview by Norik

Highly accurate
http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/

Less prone to overfitting
http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/

No need for data linearly separable

I.3 Disadvantages:
Computationally expansive

http://condor.depaul.edu/ntomuro/courses/578/notes/SVM-overview.pdf (SVM overview by Noriko Tomuro Associate professor @ DePaul University)

Can be sensitive to overfitting the hyperparameters of the model (regularization parameter
G. C. Cawley and N. L. C. Talbot, Over-fitting in model selection and subsequent selection

Long to tune
http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/

Difficulties with large datasets
https://eric.univ-lyon2.fr/~ricco/cours/slides/en/svm.pdf

Robust when there are few instances compared to number of features
https://eric.univ-lyon2.fr/~ricco/cours/slides/en/svm.pdf

I.4 Why this classifier would be a good fit for the data?
What we know about the data:
45222 instances
103 features
Continuous and binary variables

SVM is accurate which is kind of what we expect from a classifier.

II K-NN

II.1 Industrial applications:
Economic forecasting prediction
http://www.academia.edu/4607757/Application_of_K-Nearest_Neighbor_KNN_Approach_for_Predicti

Face recognition
https://www.quora.com/What-are-industry-applications-of-the-K-nearest-neighbor-algorithm?s

II.2 Advantages:

Lazy learner: No training time, no need to actualize a model when with instances
http://research.cs.tamu.edu/prism/lectures/pr/pr_l8.pdf

Adaptative behavior as it uses local information
http://research.cs.tamu.edu/prism/lectures/pr/pr_l8.pdf

Works well with big datasets
http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/

Ability to provide a range of values
https://www.quora.com/What-are-industry-applications-of-the-K-nearest-neighbor-algorithm?sl

II.3 Disadvantages:

Need to specify K
http://people.revoledu.com/kardi/tutorial/KNN/Strength%20and%20Weakness.htm

Need to determine the type of distance to use
http://people.revoledu.com/kardi/tutorial/KNN/Strength%20and%20Weakness.htm

Storage requirement
http://research.cs.tamu.edu/prism/lectures/pr/pr_l8.pdf

Need to compute the distance with each training example to predict the output -> Can be slo
http://research.cs.tamu.edu/prism/lectures/pr/pr_l8.pdf

Prone to local noise with small K
http://www.nickgillian.com/wiki/pmwiki.php/GRT/KNN

The density estimate diverge over the whole sample space
http://research.cs.tamu.edu/prism/lectures/pr/pr_l8.pdf

II.4 Why this classifier would be a good fit for the data?

What we know about the data:
45222 instances
103 features
Continuous and binary variables

Because it's a lazy learner, KNN may be desirable when data is perceived as unstructured or

III Ada Boost

III.1 Industrial Applications:

Economic predictions

```
    Soo Y. Kim, Arun Upneja, Predicting restaurant financial distress using decision tree and A

    Solder Joint inspection of chip component
    Xie Hongwei, Zhang Xianmin, Kuang Yongcong, Ouyang Gaofei, Solder Joint Inspection Method i

    Visual recognition
    https://core.ac.uk/download/pdf/21751148.pdf

    Biology, speech processing
    http://www.nickgillian.com/wiki/pmwiki.php?n=GRT.AdaBoost

III.2 Advantages:
    Few parameters to tune (unless choosing a base estimator requiring a fine tuning of many pa
    Note that the algorithm is often based on a Decision Tree Classifier
    http://www.nickgillian.com/wiki/pmwiki.php?n=GRT.AdaBoost

    Good generalization
    http://www.robots.ox.ac.uk/~az/lectures/cv/adaboost_matas.pdf

    Fast
    http://cseweb.ucsd.edu/~yfreund/papers/IntroToBoosting.pdf

III.3 Disadvantages:
    Sensitive to noisy data and outliers:
    http://www.nickgillian.com/wiki/pmwiki.php?n=GRT.AdaBoost


III.4 Why this classifier would be a good fit for the data?
    What we know about the data:
    45222 instances
    103 features
    Continuous and binary variables

    The data isnt a signal from captors. Categorical data. => Less likely to have noise

    Skewed variables have been preprocessed => No outliers

    With 45222 entry, it is worth working with a fast algorithm which has a few parameters to t

    Moreover, Schapire and Freund won the Godel Prize in 2003 for their construction of AdaBoos
```

1.6.5 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 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'
             start = time() # Get start time
             learner = learner.fit(X_train[0:sample_size], y_train[0:sample_size])
             end = time() # Get end time

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

             # TODO: Get the predictions on the test set,
             #       then get predictions on the first 300 training samples
             start = time() # Get start time
             predictions_test = learner.predict(X_test)
             predictions_train = learner.predict(X_train[0:300])
             end = time() # Get end time

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

             # TODO: Compute accuracy on the first 300 training samples
             results['acc_train'] = accuracy_score(y_train[0:300], predictions_train)

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

             # TODO: Compute F-score on the the first 300 training samples
             results['f_train'] = fbeta_score(y_train[0:300], predictions_train, beta=0.5)
```

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

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

        # Return the results
        return results
```

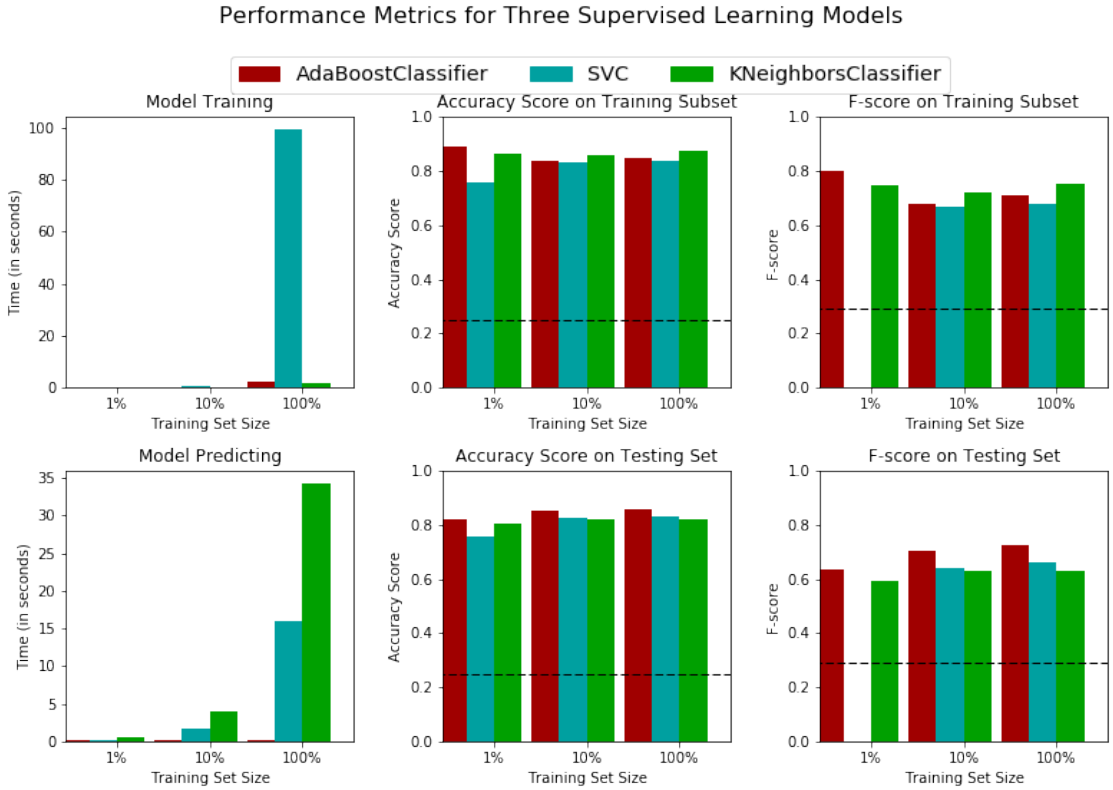
1.6.6 Implementation: Initial Model Evaluation

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 AdaBoostClassifier
         from sklearn.svm import SVC
         from sklearn.neighbors import KNeighborsClassifier

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

         # TODO: Calculate the number of samples for 1%, 10%, and 100% of the training data
         n_train = len(y_train)
         samples_1 = int(n_train*0.01)
         samples_10 = int(n_train*0.1)
         samples_100 = n_train

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

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

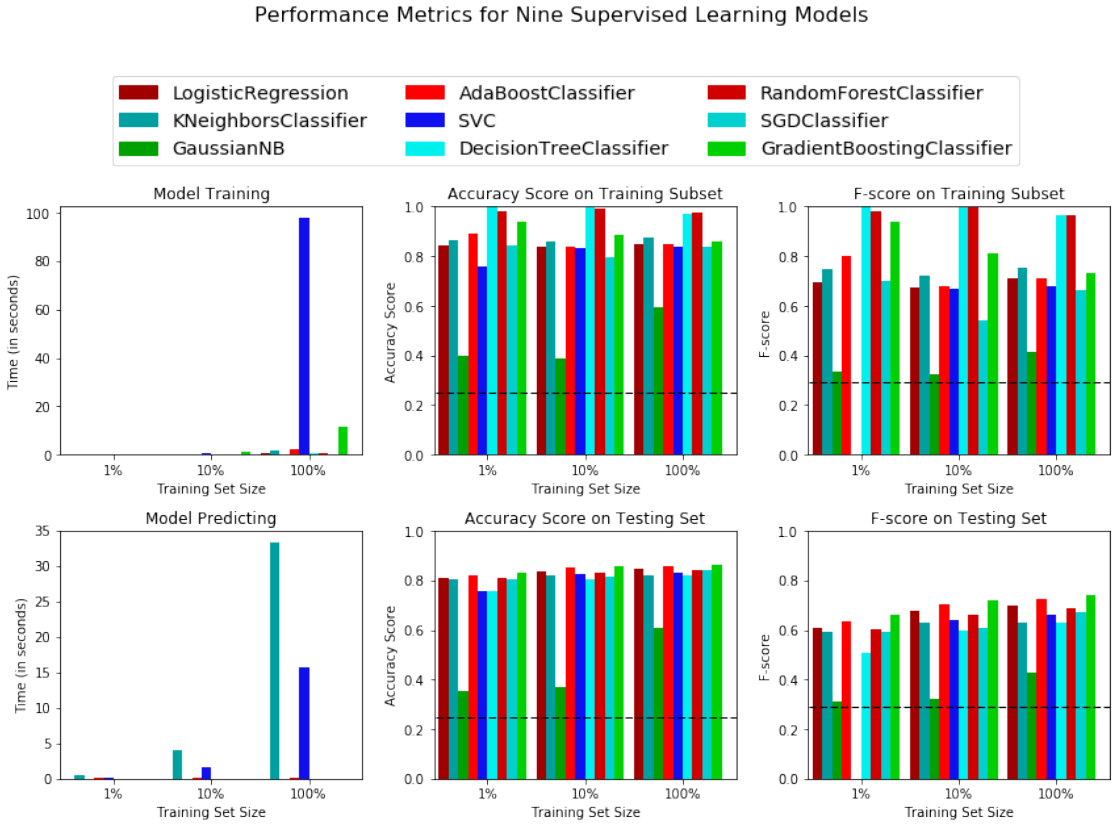
```
C:\Users\Tron\Anaconda2\lib\site-packages\sklearn\utils\validation.py:526: DataConversionWarnin
  y = column_or_1d(y, warn=True)
C:\Users\Tron\Anaconda2\lib\site-packages\sklearn\metrics\classification.py:1113: UndefinedMet
  'precision', 'predicted', average, warn_for)


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


C:\Users\Tron\Anaconda2\lib\site-packages\ipykernel\__main__.py:20: DataConversionWarning: A co


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



In [12]:
```
## Implementation: Initial Model Evaluation
# TODO: Import the three supervised learning models from sklearn
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import LogisticRegression

# TODO: Initialize the three models
clf_A = RandomForestClassifier(random_state=42)
clf_B = SVC(random_state=42)
clf_C = KNeighborsClassifier()
clf_D = GaussianNB()
clf_E = DecisionTreeClassifier(random_state=42)
clf_F = AdaBoostClassifier(random_state=42)
clf_G = GradientBoostingClassifier(random_state=42)
clf_H = SGDClassifier(random_state=42)
clf_I = LogisticRegression(random_state=42)

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the training data
n_train = len(y_train)
samples_1 = int(n_train*0.01)
samples_10 = int(n_train*0.1)
samples_100 = n_train

# Collect results on the learners
results = {}
for clf in [clf_A, clf_B, clf_C, clf_D, clf_E, clf_F, clf_G, clf_H, clf_I]:
    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_new(results, accuracy, fscore)
```

```
C:\Users\Tron\Anaconda2\lib\site-packages\ipykernel\__main__.py:20: DataConversionWarning: A co


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

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


C:\Users\Tron\Anaconda2\lib\site-packages\ipykernel\__main__.py:20: DataConversionWarning: A co


KNeighborsClassifier trained on 361 samples.
KNeighborsClassifier trained on 3617 samples.
KNeighborsClassifier trained on 36177 samples.
GaussianNB trained on 361 samples.
GaussianNB trained on 3617 samples.
GaussianNB trained on 36177 samples.
DecisionTreeClassifier trained on 361 samples.
DecisionTreeClassifier trained on 3617 samples.
DecisionTreeClassifier trained on 36177 samples.
AdaBoostClassifier trained on 361 samples.
AdaBoostClassifier trained on 3617 samples.
AdaBoostClassifier trained on 36177 samples.
GradientBoostingClassifier trained on 361 samples.
GradientBoostingClassifier trained on 3617 samples.
GradientBoostingClassifier trained on 36177 samples.
SGDClassifier trained on 361 samples.
SGDClassifier trained on 3617 samples.
SGDClassifier trained on 36177 samples.
LogisticRegression trained on 361 samples.
LogisticRegression trained on 3617 samples.
LogisticRegression trained on 36177 samples.
```



1.7 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.

1.7.1 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: Your answer should include discussion of the metrics, prediction/training time, and the algorithm's suitability for the data.

Answer: As identifying someone that does not make more than $50,000 as someone who does would be detrimental to CharityML, it is more important to choose an algorithm with a good F-score than an algorithm with a good accuracy.

AdaBoost obtains better F-score than SVM and K-NN. Moreover SVM is very slow on the training stage and K-NN is really slow on the predicting stage. As AdaBoost has only a few

parameters to tune compared to SVM, it will be much more faster to search for the AdaBoost optimal parameters than for the SVM optimal parameters on this dataset (45222x103).

For those reasons AdaBoost is more likely to obtain the best performance in the tuning stage.

1.7.2 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 or technical jargon, such as describing equations or discussing the algorithm implementation.*

Answer: The model that has been chosen to make prediction is the AdaBoost classifier. It is a combination of several simple classifiers. Here is described how the algorithm globally works:

Training stage: a simple DecisionTree is trained on the training dataset. Data that are misclassified are given higher/louder weights and data which are correctly classified are given lower/lighter weights. A new Decision Tree is train and new weights are given. Those 2 steps are repeated N times.

Predicting stage: Each prediction is a combination of the results of the differents Decision Trees. Those Decision Trees' results are weighted according to their distance with the example to classify.

1.7.3 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 [16]: # TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
         from sklearn.metrics import make_scorer
         from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import StratifiedShuffleSplit

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

         # TODO: Create the parameters list you wish to tune
         n_estimators_range = np.linspace(115, 120, 6,dtype=int)
         learning_rate_range = np.linspace(1.45, 1.55, 6)
         parameters = dict(n_estimators=n_estimators_range, learning_rate=learning_rate_range)
```

```
        cv = StratifiedShuffleSplit(n_splits=20, test6_size=0.2, random_state=42)

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

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

        # TODO: Fit the grid search object to the training data and find the optimal paramete
        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, predictio
        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_pred
```

```
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: 0.8633
Final F-score on the testing data: 0.7331
```

1.7.4 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	Benchmark Predictor	Unoptimized Model	Optimized Model
Accuracy Score	0.2478	0.8576	0.8633

Metric	Benchmark Predictor	Unoptimized Model	Optimized Model
F-score	0.3501	0.7246	0.7331

Results: Answer: The optimized scores are significantly better than the unoptimized ones. There's a gain of nearly 1% for each score which could be kind of equivalent to 450 entry.

Naive predictor results: Accuracy score: 0.2478, F-score: 0.2917 Both the unoptimized and optimized models are better than the naive predictor. However, the Naive predictor has higher F-score than Accuracy, which is not the case with the trained predictors.

1.8 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.

1.8.1 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 think features related to money (capital_gain, capital_loss) are highly correlated to income.

It is likely to earn to have a higher salary for an older worker, so the age feature may be important as well.

Features charactirizing the level of avancement (education_num, education_level) in studies may be correlated to income too, with a slight advantage to education_num which can take into account studies during the professional life.

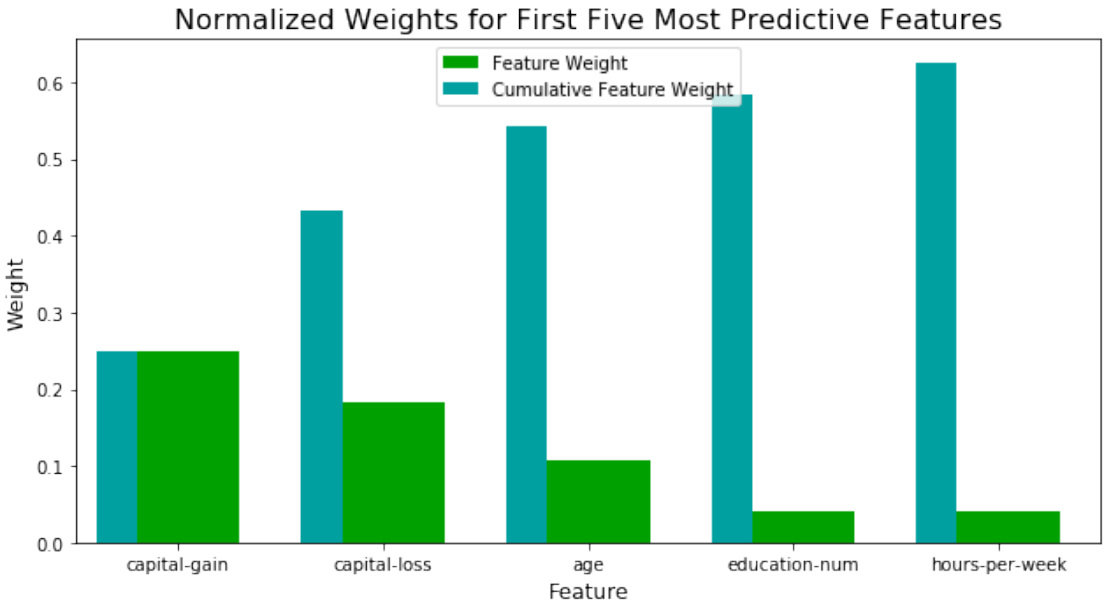
People who work less are not likely to have high income, so the hours_per_week feature may be important too.

1.8.2 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 [17]: # TODO: Import a supervised learning model that has 'feature_importances_'

         # TODO: Train the supervised model on the training set
         model = AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
                                    learning_rate=1.53, n_estimators=120, random_state=42)

         model = model.fit(X_train, y_train)

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

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



1.8.3 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: Those are the same features than the one I have chosen earlier.

1.8.4 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 [18]: # Import functionality for cloning a model
         from sklearn.base import clone

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

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

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

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

```
Final Model trained on full data
------
Accuracy on testing data: 0.8633
F-score on testing data: 0.7331

Final Model trained on reduced data
------
Accuracy on testing data: 0.8367
F-score on testing data: 0.6813
```

1.8.5 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: With a loss of 5% on the F-score and nearly 3% on the accuracy score (nearly 1350 entry), I would not use that kind of feature selection. If training time was a factor, I would use PCA to reduce dimension while keeping the most information possible.

> **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.