

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) (<https://archive.ics.uci.edu/ml/datasets/Census+Income>). The dataset 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 'fnlwtgt' 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 [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	occupation	relationship	race	sex	capital-gain
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	21

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 = data['age'].count()

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

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

# TODO: Percentage of individuals whose income is more than $50,000
greater_percent = float(n_greater_50k)*100/n_records

# Print the results
print "Total number of records: {}".format(n_records)
print "Individuals making more than $50,000: {}".format(n_greater_50k)
print "Individuals making at most $50,000: {}".format(n_at_most_50k)
print "Percentage of individuals making more than $50,000: {:.2f}%".format(greater_percent)
```

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

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, Trinidad&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.

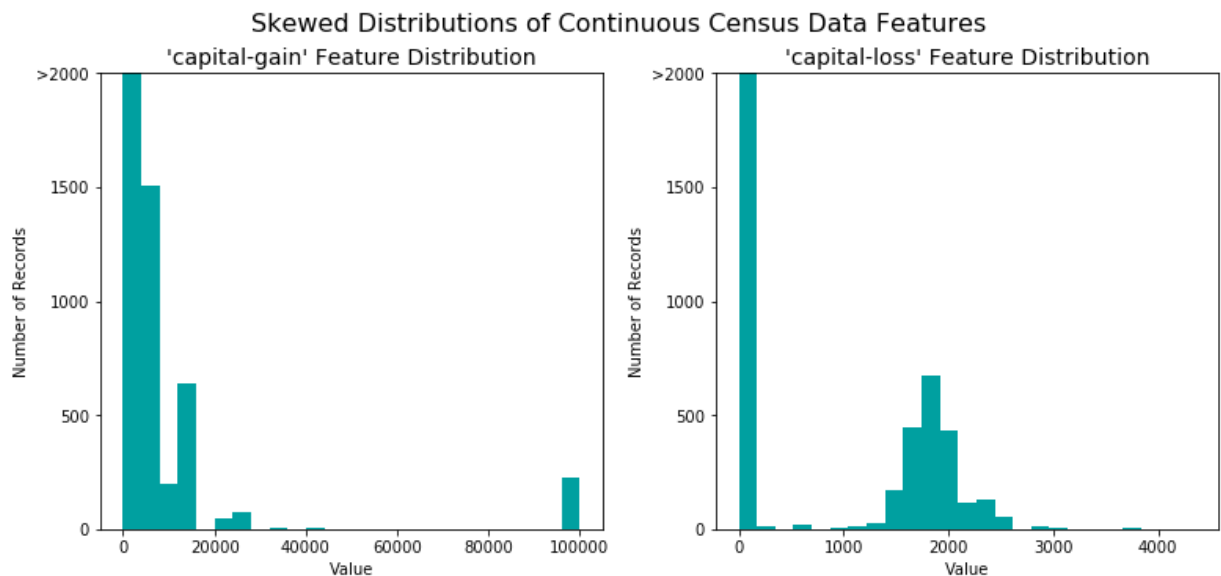
Transforming Skewed Continuous Features

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 [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)
```



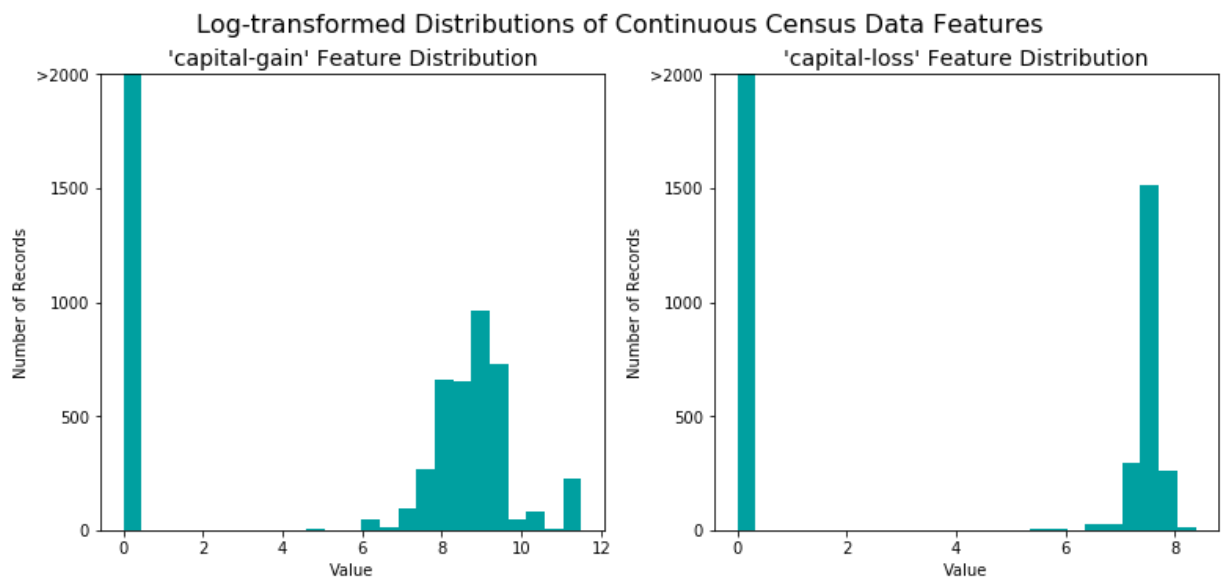
For highly-skewed feature distributions such as 'capital-gain' and 'capital-loss', it is common practice to apply a [logarithmic transformation](https://en.wikipedia.org/wiki/Data_transformation_(statistics)) on the data so that the very large and very small values do not negatively affect the performance 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 [4]: # 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 exemplified 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>) for this.

```
In [5]: # 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-w

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

# 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	race	sex
0	0.301370	State-gov	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family	White	M
1	0.452055	Self-emp-not-inc	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband	White	M
2	0.287671	Private	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family	White	M
3	0.493151	Private	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband	Black	M
4	0.150685	Private	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wife	Black	Fem

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_raw' 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()

from sklearn.preprocessing import LabelEncoder

#Encoder initialization

encoder = LabelEncoder()

#The One-Hot encoding

features_final = pd.get_dummies(features_raw)

# TODO: Encode the 'income_raw' data to numerical values
income = encoder.fit_transform(income_raw)

# 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.
['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week', 'workclass_Federal-gov', 'workclass_Local-gov', 'workclass_Private', 'workclass_Self-emp-inc', 'workclass_Self-emp-not-inc', 'workclass_State-gov', 'workclass_Without-pay', 'education_level_10th', 'education_level_11th', 'education_level_12th', 'education_level_1st-4th', 'education_level_5th-6th', 'education_level_7th-8th', 'education_level_9th', 'education_level_Assoc-acdm', 'education_level_Assoc-voc', 'education_level_Bachelors', 'education_level_Doctorate', 'education_level_HS-grad', 'education_level_Masters', 'education_level_Preschool', 'education_level_Prof-school', 'education_level_Some-college', 'marital-status_Divorced', 'marital-status_Married-AF-spouse', 'marital-status_Married-civ-spouse', 'marital-status_Married-spouse-absent', 'marital-status_Never-married', 'marital-status_Separated', 'marital-status_Widowed', 'occupation_Adm-clerical', 'occupation_Armed-Forces', 'occupation_Craft-repair', 'occupation_Exec-managerial', 'occupation_Farming-fishing', 'occupation_Handlers-cleaners', 'occupation_Machine-op-inspct', 'occupation_Other-service', 'occupation_Priv-house-serv', 'occupation_Prof-specialty', 'occupation_Protective-serv', 'occupation_Sales', 'occupation_Tech-support', 'occupation_Transport-moving', 'relationship_Husband', 'relationship_Not-in-family', 'relationship_Other-relative', 'relationship_Other', 'relationship_Wife', 'relationship_With-parents']
```

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_

        # 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 performance would be appropriate. Additionally, identifying someone that *does not* make more than \$50,000 as someone who does would be detrimental to *CharityML*, since they are looking to find individuals willing to donate. Therefore, a model's ability to precisely predict those that make more than \$50,000 is *more important* than the model's ability to **recall** those individuals. We can use **F-beta score** as a metric that considers both precision and recall:

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

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

$$[\text{True Positives}/(\text{True Positives} + \text{False Positives})]$$

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

$$[\text{True Positives}/(\text{True Positives} + \text{False Negatives})]$$

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 Performance

- 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.

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($\text{True Positives}/(\text{True Positives} + \text{False Positives})$) 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($\text{True Positives}/(\text{True Positives} + \text{False Negatives})$) in this setting becomes 1 as we have no False Negatives.

```

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

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

# TODO: Calculate accuracy, precision and recall

from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import fbeta_score

income_pred = income.apply(lambda x:1) #

#Calculating True Positive
#TP = sum(map(lambda x,y: 1 if x==1 and y ==1 else 0, income, income_pred))
#Calculating False Positive
#FP= sum(map(lambda x,y:1 if x==0 and y == 1 else 0, income, income_pred))
#calculating False Negative
#FN=sum(map(lambda x,y: 1 if x==1 and y == 0 else 0, income, income_pred))

accuracy = float(n_greater_50k)/float(n_at_most_50k +n_greater_50k)
#print accuracy (used to test my calculation)

recall = float(n_greater_50k)/float(n_greater_50k)

#print recall(used to view my result of recall during the implementation)

precision =float(n_greater_50k)/float(n_at_most_50k+n_greater_50k)

# TODO: Calculate F-score using the formula above for beta = 0.5 and correct value
# HINT: The formula above can be written as (1 + beta**2) * (precision * recall) / (1 + beta**2 + recall)
beta=0.5
fscore = (1 + beta **2) * (precision*recall)/((precision*beta**2) + recall)

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

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

```

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) (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:** Three of the supervised learning models I will consider appropriate for this problem: Gaussian Naive Bayes, Support Vector Machines and Decision Trees.

Support Vector Machines (SVM)

Real World Application

SVM algorithm has been widely applied in biological and other sciences. Mostly to classify proteins with up to 90% of the compounds classified correctly.[1]

The strength of the Model and when it performs well

When we need better separation of the classes SVM is a good choice. SVM performs well on data sets that have many attributes, even if there are very few cases on which to train the model. There is no upper limit on the number of attributes; the only constraints are those imposed by hardware.[2] SVM can be versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.[3]

The weaknesses of the model, and when it performs poorly

SVMs are binary classifiers, you will need to write a little more code to use an SVM on a problem with more than two classes. The choice of the Kernel also presents some challenges. Also, from a practical point of view perhaps the most serious problem with SVMs is the high algorithmic complexity and extensive memory requirements of the required quadratic programming in large-scale tasks.[4] Using SVM, training a classifier on a large number of documents can involve a large amount of computing resources. [5]

Why this model is a good candidate for this problem

SVMs can work well on text classification; we treat each document as a vector with tens of thousands of features.[6] This data has features that can clearly define the margins that makes it possible to delineate the income levels and thus able to classify them to a high degree of accuracy.

References:

- [1].(https://en.wikipedia.org/wiki/Support_vector_machine
(https://en.wikipedia.org/wiki/Support_vector_machine))
- [2] https://docs.oracle.com/cd/B28359_01/datamine.111/b28129/algo_svm.htm#CHDDJFDJ
(https://docs.oracle.com/cd/B28359_01/datamine.111/b28129/algo_svm.htm#CHDDJFDJ)
- [3] <http://scikit-learn.org/stable/modules/svm.html> (<http://scikit-learn.org/stable/modules/svm.html>)
- [4] Horváth (2003) in Suykens et al. p 392
- [5,6].(Harrington, Peter. Machine Learning in Action, New York: Manning Publications Co. 2012)

Gaussian Naïve Bayes (GaussianNB)

One real-world application

Gaussian Naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering.[1]

The strengths of the model; when does it perform well

Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. An advantage of naive Bayes is that it only requires a small number of training data to estimate the parameters necessary for classification.[2]

The weaknesses of the model; when does it perform poorly

Gaussian Naive Bayes is known to be a bad estimator, therefore the probability outputs can be misleading.[3]

Why the model is a good candidate for the problem, given what you know about the data

In our example we have a large data set and not many features, GaussianNB is a good algorithm for these cases.

References:

- [1] (http://scikit-learn.org/stable/modules/naive_bayes.html (http://scikit-learn.org/stable/modules/naive_bayes.html))
- [2] (https://en.wikipedia.org/wiki/Naive_Bayes_classifier
(https://en.wikipedia.org/wiki/Naive_Bayes_classifier))
- [3] (http://scikit-learn.org/stable/modules/naive_bayes.html (http://scikit-learn.org/stable/modules/naive_bayes.html))

Decision Tree

One real-world application in industry where the model can be applied

Decision tree learning has been applied to problems such as learning to classify medical patients by their disease. [1]

The strengths of the model; when does it perform well

-Simple to understand and to interpret. Trees can be visualised. -Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values. -The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree. Able to handle both numerical and categorical data. Other techniques are usually specialised in analysing datasets that have only one type of variable. -Able to handle multi-output problems. -Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret. -Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model. Performs well even if its assumptions are somewhat violated by the true model from which the data were generated. [2]

The weaknesses of the model; when does it perform poorly

-Prone to overfitting: Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. -Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble. -The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement. -There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems. -Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.[3]

What makes this model a good candidate for the problem, given what you know about the data

For the data we did not need to do data normalization also the power of Decision trees in classification makes it a good choice in determining which members of the data set earn more than 50k.

References

[1] (Mitchell, Tom, M. , Machine Learning, Mcgraw-Hill, March 1, 1997)

[2] (<http://scikit-learn.org/stable/modules/tree.html> (<http://scikit-learn.org/stable/modules/tree.html>))

[3] (<http://scikit-learn.org/stable/modules/tree.html> (<http://scikit-learn.org/stable/modules/tree.html>))

**

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

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

    # TODO: Get the predictions on the test set(X_test),
    # then get predictions on the first 300 training samples(X_train) using
    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_score
    results['f_train'] = fbeta_score(y_train[:300], predictions_train, beta=0.5)

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

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

    # Return the results
    return results

```

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 [12]: # TODO: Import the three supervised learning models from sklearn

from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

# TODO: Initialize the three models
clf_A = SVC(random_state=0)
clf_B = GaussianNB()
clf_C = DecisionTreeClassifier(random_state=0)

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the training data
# HINT: samples_100 is the entire training set i.e. len(y_train)
# HINT: samples_10 is 10% of samples_100
# HINT: samples_1 is 1% of samples_100

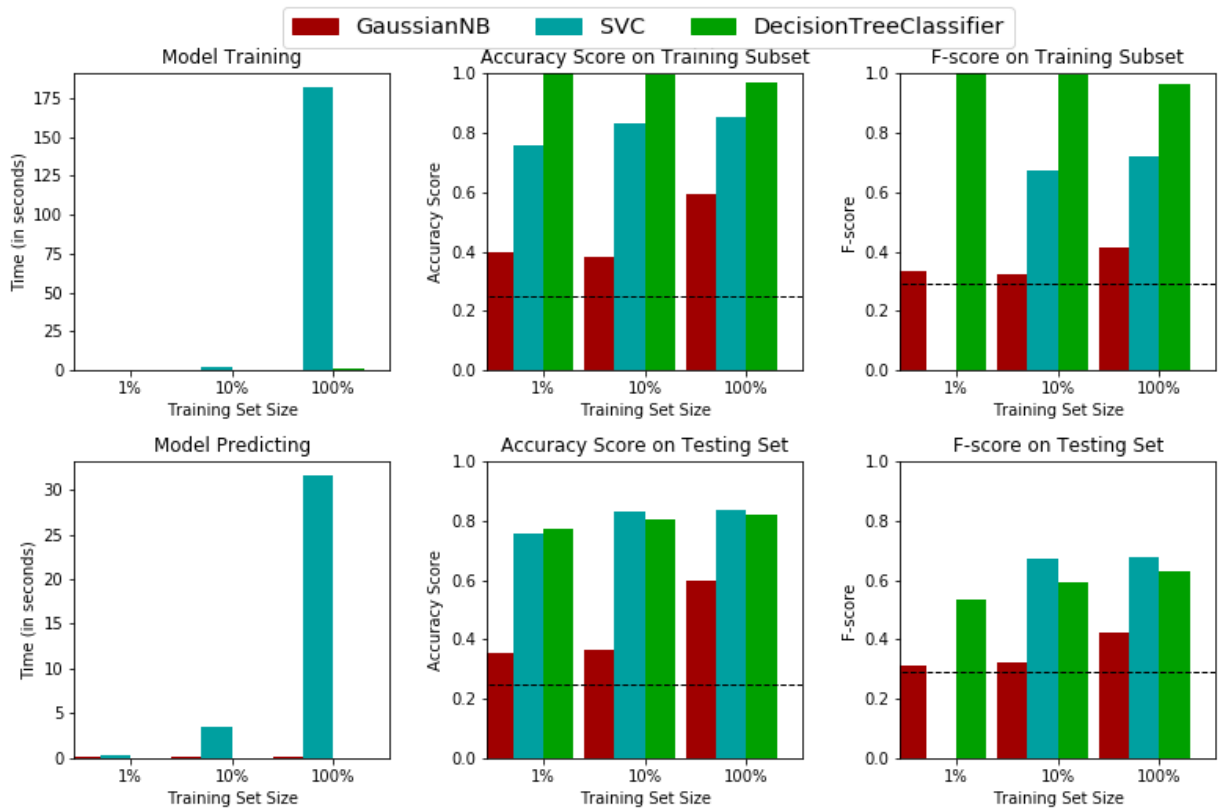
samples_100 = len(X_train)
samples_10 = len(X_train)/10
samples_1 = len(X_train)/100

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

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

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

Performance Metrics for Three Supervised Learning Models



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:**

The F-score is a single measure of classification procedure's usefulness; the larger the training set size the greater was the Support Vector Machine(SVM) results, this could indicate it might be more useful of the three for analyzing this data. The accuracy results for the testing set data also show an increase in the accuracy score as the training set size increases using SVM usually in the 80% area the DecisionTreeClassifier was close to these results however SVM was consistent in outperforming as training size increases.

. Upon doing Analysis of the three algorithm: GaussianNB, Support Vector Machine and DecisionTreeClassifier it is clear GaussianNB and DecisionTreeClassifier are training times are virtually the same however Support Vector Machine was highest by a relatively large magnitude. The Prediction time showed similar results to training times with Support Vector Machine being the highest and GaussianNB and DecisionTreeClassifier being lower and similar in magnitude.

For Training subsets DecisionTreeClassifier was the most consistent in being accurate and having a higher F-score(usually near or at 100%); however given the large population that might be needed to solicit donation from a more robust Algorithm for increasing sample size maybe needed. As a result the Support Vector Machine will prove more suitable for this data.

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:** Support Vector Machine(SVM)

If we have a situation where we have two groups of data, and the data points are separated enough that you could draw a straight line on the figure with all the points of one class on one side of the line and all the points of the other class on the other side of the line, we say the data is linearly separable. The line used to separate the dataset is called a separating hyperplane. But, if we have a dataset with three dimensions, we need a plane to separate the data; and if we have data with 1024 dimensions, we need something with 1023 dimensions to separate the data. What do you call something with 1023 dimensions, this is called a hyperplane. The hyperplane is our decision boundary. Everything on one side belongs to one class, and everything on the other side belongs to a different class. We'd like to find the point closest to the separating hyperplane and make sure this is as far away from the separating line as possible. This is known as margin. We want to have the greatest possible margin, because if we made a mistake or trained our classifier on limited data, we'd want it to be as robust as possible. The points closest to the separating hyperplane are known as support vectors. Now that we know that we're trying to maximize the distance from the separating line to the support vectors, we need to find a way to optimize this problem.[1]We are able to do this optimization by applying some mathematical functions called 'kernels'. A Kernel takes two inputs and spits out how similar they are.[2]

How this model will work for the data for CharityML; so we have the census data and we need to identify who makes over fifty thousand dollars and could become potential donors. The Support Vector Machine(SVM) will be used to identify potential donors. We will train the SVM on a portion of

the data, this will be done by finding the hyperplane that will classify the data into two classes. This is where the mathematical function we discussed earlier comes in. With implementing correct mathematical equations we can maximize the margin between the two classes and hence maximize the separability of the classes. The equations used determine what will be the Support Vectors and these vector points constitute how this hyperplane is defined; and thus with this classifier in place data points not in the training data can be accurately predicted to be correctly assigned to a fair degree to their respective class.[3]

References

[1]Harrington, Peter. Machine Learning in Action, New York: Manning Publications Co., 2012

[2]https://en.wikipedia.org/wiki/Kernel_method (https://en.wikipedia.org/wiki/Kernel_method)

[3]SERGIOS THEODORIDIS Department of Informatics and Telecommunications University of Athens Greece and KONSTANTINOS KOUTROUMBAS Institute of Space Applications & Remote Sensing National Observatory of Athens Greece. PATTERN RECOGNITION, SECOND EDITION , Academic Press, San Diego., 2003

**

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

Answer:

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 [14]: # TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary libraries

from sklearn.grid_search import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.metrics import fbeta_score

# TODO: Initialize the classifier
clf = SVC(random_state =0)

# TODO: Create the parameters list you wish to tune, using a dictionary if needed
# HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, value2]}
parameters = {'C':[1, 10, 100], 'kernel':['rbf', 'linear']}

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

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

# TODO: Fit the grid search object to the training data and find the optimal parameters
grid_fit = grid_obj.fit(X_train, y_train)

# Get the estimator
best_clf = grid_fit.best_estimator_

# Make predictions using the unoptimized and model
predictions = (clf.fit(X_train, y_train)).predict(X_test)
best_predictions = best_clf.predict(X_test)

# Report the before-and-afterscores
print "Unoptimized model\n-----"
print "Accuracy score on testing data: {:.4f}".format(accuracy_score(y_test, predictions))
print "F-score on testing data: {:.4f}".format(fbeta_score(y_test, predictions, beta=0.5))
print "\nOptimized Model\n-----"
print "Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test, best_predictions))
print "Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, beta=0.5))
```

Unoptimized model

Accuracy score on testing data: 0.8371

F-score on testing data: 0.6745

Optimized Model

Final accuracy score on the testing data: 0.8391

Final F-score on the testing data: 0.6791

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	Benchmark Predictor	Unoptimized Model	Optimized Model
Accuracy Score	0.2478	0.8371	0.8391
F-score	0.2917	0.6745	0.6791

**Answer:*

From the scores obtain from model tuning it can be seen that both the unoptimized and optimized models perform much better than the naive predictor benchmarks. For accuracy score the optimized model performed at about 0.23% better than unoptimized model. For the F-score the Optimized model improved a by a little over 0.6%. With more fine-tuned values for the parameters the Optimized model could improve, however with SVM the time taken to process the data drastically increase and hence a more intricate fine tuning could prove counter-productive. *

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:**

1.Education Level - Individuals with a high level of education and are able to find jobs suitable to their edcation level will more likely earn higher incomes than the general population 2.Age - Experience usually correlates with higher income, younger people will most likely have less experience and will this earn lower incomes than older individuals. 3.Hours per Week- More hours worked per week usually have a positive correlltion with higher income, particularly if the individuals being compared are in similar income brackets. 4.Capital-gains- Individuals who are subjected to capital gain taxes are usually individuals with high income and investable assets; they would make

good candidates for donation. 5. Workclass: Working in the private sector usually allows for higher disposable income than with being in government where bonuses are usually not as common. These individuals who can earn more than their fixed salary could be good candidates for making donation. **

Implementation - Extracting Feature Importance

Choose a scikit-learn supervised learning algorithm that has a `feature_importance_` attribute available 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:

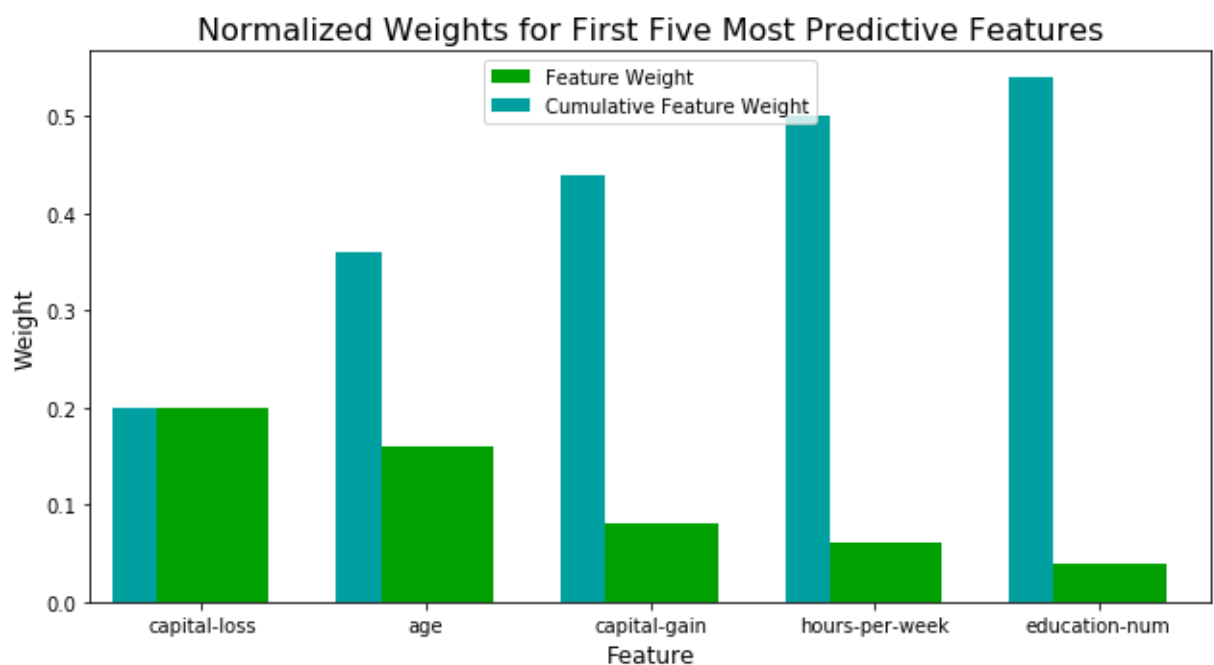
- Import a supervised learning model from sklearn if it is different from the three used earlier.
- Train the supervised model on the entire training set.
- Extract the feature importances using `feature_importances_`.

```
In [15]: # 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_train)
model = AdaBoostClassifier(random_state=0)
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:**

-Four of the features I selected was similar to the feature importance implementation. -The visulatization accounted for the factors I would consider are important in what constitutes good metrics to determine good income. If an individual earns good income they will more likely be able to have disposable income for making donations. **

Feature Selection

How does a model perform if we only use a subset of all the available features in the data? With less featizures 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 [16]: # 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])]]
X_test_reduced = X_test[X_test.columns.values[(np.argsort(importances)[::-1])[:5]]]

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

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

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

Final Model trained on full data
-----
Accuracy on testing data: 0.8391
F-score on testing data: 0.6791

Final Model trained on reduced data
-----
Accuracy on testing data: 0.8051
F-score on testing data: 0.5691
```

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:**

-Reducing the data using only five features resulted in a reduction in both the Accuracy and the F-score by 4% and 16% respectively. -Using SVM as the preferred algorithm it would be advisable to use the reduced data; this is because the time taken to process using all the data could prove counter-productive. However, another algorithm may make training with the full dataset feasible**

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.

