

# Finding\_Donors

May 27, 2018

## 1 Finding Donors for *CharityML*

Welcome to the first project of the Machine Learning Basics 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:** I am using Python. 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.

### 1.1 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 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](#). 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.

## 1.2 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
data.head(1)
```

```
Out[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			

### 1.2.1 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 = data.shape[0]

# TODO: Number of records where individual's income is more than $50,000
```

```

n_greater_50k = (data['income'] == '>50K').sum()

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

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

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

```

```

Total number of records: 45222
Individuals making more than $50,000: 11208
Individuals making at most $50,000: 34014
Percentage of individuals making more than $50,000: 24.78439697492371%

```

## **\*\* Featureset Exploration \*\***

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

## 1.3 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.

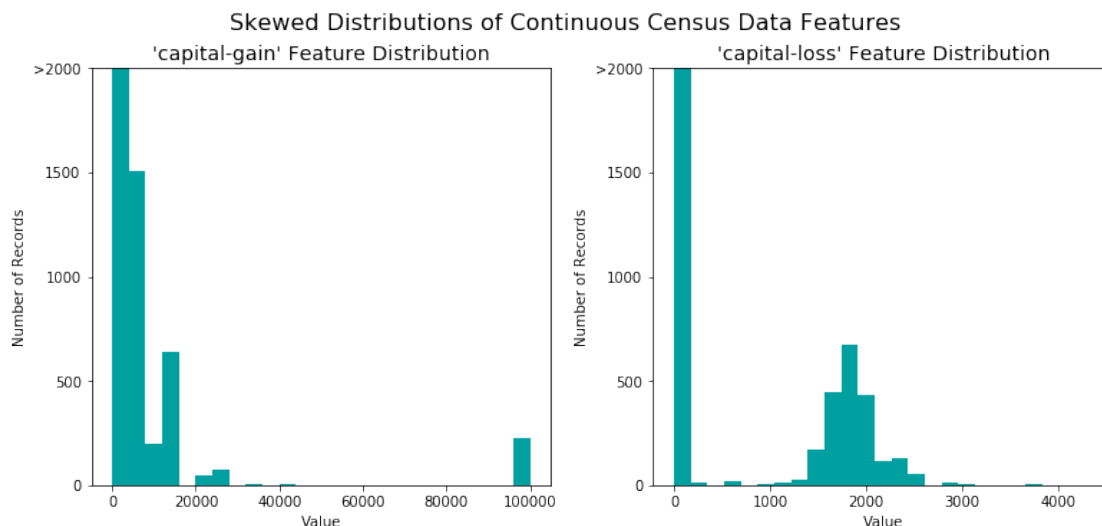
### 1.3.1 Transforming Skewed Continuous Features

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)
```

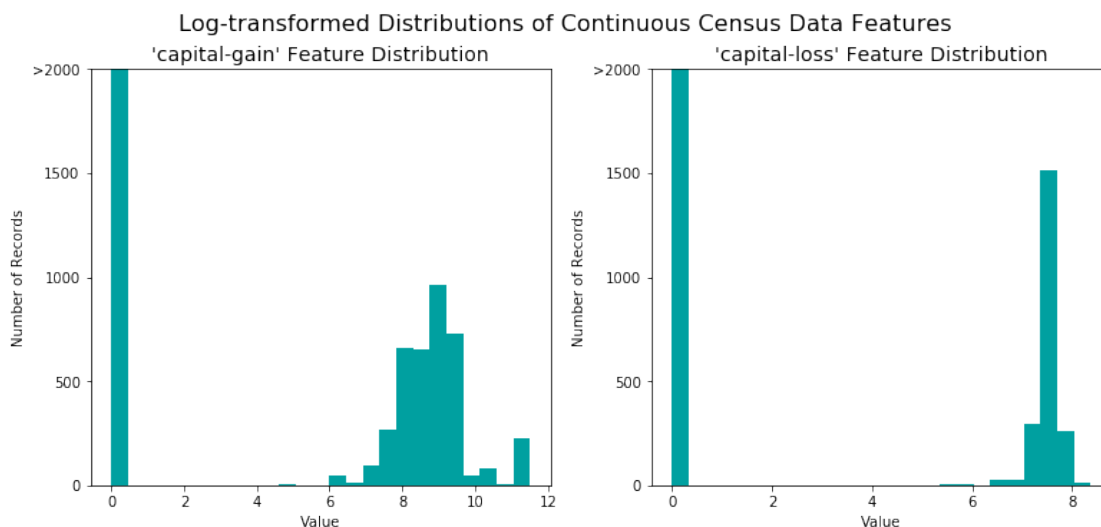


For highly-skewed feature distributions such as 'capital-gain' and 'capital-loss', it is common practice to apply a logarithmic transformation on the data so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation 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_log_transformed = pd.DataFrame(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)
```



### 1.3.2 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 exemplified 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() # default=(0, 1)
numerical = ['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week']

features_log_minmax_transform = pd.DataFrame(data = features_log_transformed)
```

```
features_log_minmax_transform[numerical] = scaler.fit_transform(features_log_transform)

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

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

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

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

### 1.3.3 Implementation: Data Preprocessing

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

```
| someFeature | | someFeature_A | someFeature_B | someFeature_C |
:-: | :-: | | :-: | :-: | :-: |
0 | B | | 0 | 1 | 0 |
1 | C | —> one-hot encode —> | 0 | 0 | 1 |
2 | A | | 1 | 0 | 0 |
```

Additionally, as with the non-numeric features, we need to convert the non-numeric target label, 'income' to numerical values for the learning algorithm to work. Since there are only two possible categories for this label (“<=50K” and “>50K”), we can avoid using one-hot encoding and simply encode these two categories as 0 and 1, respectively. In code cell below, you will need to implement the following: - Use `pandas.get_dummies()` to perform one-hot encoding on the 'features\_log\_minmax\_transform' data. - Convert the target label 'income\_raw' to numerical entries. - Set records with “<=50K” to 0 and records with “>50K” to 1.

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

# TODO: Encode the 'income_raw' data to numerical values
income = income_raw.apply(lambda x: 1 if x == '>50K' else 0)

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

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

103 total features after one-hot encoding.

['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week', 'workclass\_ Federal

### 1.3.4 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_final,
                                                    income,
                                                    test_size = 0.2,
                                                    random_state = 0)

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

Training set has 36177 samples.

Testing set has 9045 samples.

c:\users\juzer\appdata\local\programs\python\python36-32\lib\site-packages\sklearn\cross\_valid  
 "This module will be removed in 0.20.", DeprecationWarning)

## 1.4 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*.

### 1.4.1 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 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<sub>0.5</sub> score** (or F-score for simplicity).

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

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

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

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

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

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

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



weighted average(harmonic mean) of the precision and recall scores. This score can range from 0 to 1, with 1 being the best possible F1 score(we take the harmonic mean as we are dealing with ratios).

### 1.4.2 Question 1 - Naive Predictor Performace

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

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

**\*\* HINT: \*\***

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

```
In [8]: # Counting the ones as this is the naive case. Note that 'income' is the 'income_raw' fo
        TP = np.sum(income)

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

        # TODO: Calculate F-score using the formula above for beta = 0.5 and correct values fo
        #initiating beta value to be used in fscore formula
        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}]"
```

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

### 1.4.3 Supervised Learning Models

The following are some of the supervised learning models that are currently available in [scikit-learn](#) that you may choose from: - Gaussian Naive Bayes (GaussianNB) - Decision Trees - Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting) - K-Nearest Neighbors (KNeighbors) - Stochastic Gradient Descent Classifier (SGDC) - Support Vector Machines (SVM) - Logistic Regression

### 1.4.4 Question 2 - Model Application

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

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

**\*\* HINT: \*\***

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

**Answer:** The models I have chosen are following:

**Gradient Boosting Classifier (GBC):** **\*\* Usage: \*\*** GBC can be used in the field of [learning to rank](#). The commercial web search engines [Yahoo](#) and [Yahndex](#) use variants of gradient boosting in their machine learned ranking engines.

**Strengths:** The strengths of this model are that it builds new trees which complement the already built trees. The new trees which will be built will help to correct errors in the previously built trees. This can produce highly accurate results with less trees. Its quite powerful and also better than random forest classifier.

**\*\* Weakness: \*\*** The weakness of this model is that its time consuming, harder to run parallelly, prone to overfitting and does not scale well.

**\*\* Why use this?: \*\*** Its a good candidate because it produces highly accurate results which we need.

Reference [here](#).

---

**AdaBoost Classifier: Usage:** In Industries, this algorithm is used for binary classification problems of face detection where the algorithm has to identify whether the portion of an image is face or background. It also had alot of practical success in field of biology, computer vision and speech processing.

**Strengths:** Unlike SVM or other power classifiers, AdaBoost can achieve similar classification result with much less tweaking or settings. Its fast and also less prone to over fit.

**Weakness:** Noisy data and outleirs in the dataset can negatively impact the performance of the algorithm.

**\*\* Why use this?: \*\*** Since we have a good training data points and clean dataset we will be able to perform multiple training iterations to maximize overall accuracy on the unseen testing data.

Reference [here](#)

---

**Stochastic Gradient Descent Classifier (SGDC)** **Usage:** It is used in Text classification and Natural Language Processing (NLP)

**Strengths:** Very efficient, easy to implement and provides lots of parameter tuning.

**Weakness:** A number of hyperparameters are required and is sensitive to feature scaling.

**\*\* Why use this?: \*\*** - We have greater than 50 sample size, enough to train with. - Labelled Data - Predicting category. (Classification)

Reference [here](#)

### 1.4.5 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 [9]: # TODO: Import two metrics from sklearn - fbeta_score and accuracy_score
        from sklearn.metrics import fbeta_score, accuracy_score

        def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
            '''
            inputs:
            - learner: the learning algorithm to be trained and predicted on
            - sample_size: the size of samples (number) to be drawn from training set
            - X_train: features training set
            - y_train: income training set
            - X_test: features testing set
            - y_test: income testing set
            '''

            results = {}

            # TODO: Fit the learner to the training data using slicing with 'sample_size' using
            start = time() # Get start time
            learner = 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 .pre
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

```

### 1.4.6 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 [10]: # TODO: Import the three supervised learning models from sklearn
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from sklearn.linear_model import SGDClassifier

# TODO: Initialize the three models
clf_A = AdaBoostClassifier(random_state=0)

```

```

clf_B = GradientBoostingClassifier(random_state=0)
clf_C = SGDClassifier(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 (ensure to set the count of the values to be `
# HINT: samples_1 is 1% of samples_100 (ensure to set the count of the values to be `
samples_100 = int(len(y_train))
samples_10 = int(len(y_train) * 0.1)
samples_1 = int(len(y_train) * 0.01)

# 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)

```

```

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.

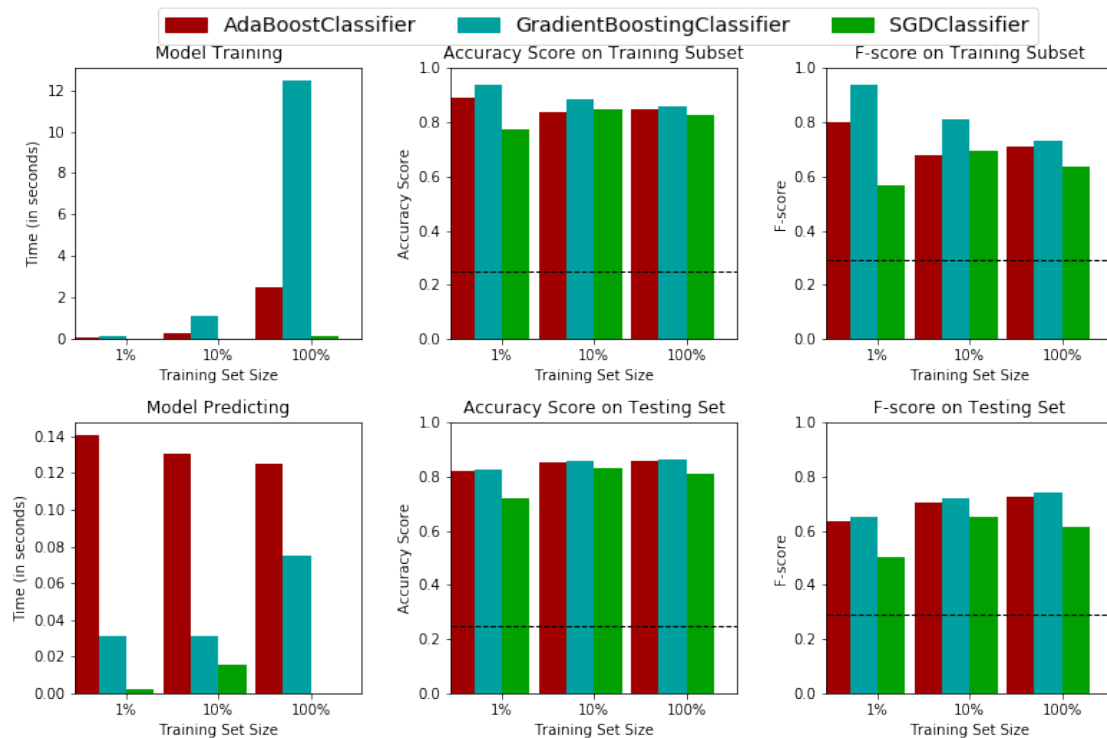
```

```

c:\users\juzer\appdata\local\programs\python\python36-32\lib\site-packages\sklearn\linear_model:
    "and default tol will be 1e-3." % type(self), FutureWarning)

```

## Performance Metrics for Three Supervised Learning Models



## 1.5 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.

### 1.5.1 Question 3 - Choosing the Best Model

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

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

**Answer:** Out of the 3 Classification Algorithms I chose, the best is **GradientBoostingClassifier (GBC)** Hands Down.

GBC produces highest accuracy and f-score both in training set and testing set. It also doesn't overfit the data since we have a good accuracy score in training set of approx 85% and we see similar accuracy score in testing set too which means it does a very good job of predicting who makes more than \$50k with given set of features. Same argument I can make about underfitting, which it doesn't do at all. Talking about the f-score now, it is nearly around 80% which is highest among other algorithms. The higher the f-score, the greater the models prediction and recall which are surely important for our data.

As far as time goes, GBC does take the longest time to train but it does a real nice and fast job when it comes to prediction. AdaBoost Classifier takes the shortest time to train but when it comes to prediction, it takes the longest. Though, it also does a pretty well job in accuracy and f-score both in training and testing set but GBC takes is better in accuracy and f-score and also when it comes to model prediction time. The time taken by GBC while training the model does go high as our training set increases. It's a little trade-off we make here if we want better prediction results.

### 1.5.2 Question 4 - Describing the Model in Layman's Terms

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

**\*\* HINT: \*\***

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

**Answer:** You can think of Machine Learning (ML) as a methodology of statistical guessing, of which there are many algorithms. GBC is a type of **Tree Model**.

A Tree model is, in layman's terms, like the Twenty Questions guessing game. The guesser might have questions like "Is it bigger than a bread box?", "Is it alive?", etc. The size or lifeness of the thing being guessed at is a "feature". By winnowing down what is likely or unlikely based on these questions, you end up with a likely (but possibly wrong) answer. Part of the strategy in 20 questions is to order the questions correctly: the first few questions should be broad, so as to eliminate large number of possibilities. The last few questions should be more specific to hone in on the "best" possible answer.

Now, what happens when a Tree ML is trained on the data set, the algorithm tries to come up with a set of "questions" that are 'optimal'. Unfortunately, there is no perfect solution. So, there are different strategies to try to build the Tree Model. GBC is one of the Tree models.

**GBC:** Imagine 20 teams (trees). A boss at the top, then subordinates, then more subordinates, and so on. Team members are explanatory variables. Assume,  $Trees = 20$  and  $Depth$  (number of members in each team) = 5. So each team will have 5 members, and total members = 100. We give them a book to read, and then they will have to answer 20 questions (Number of observations in our data). Assume they have binary answers: Yes or No (in our case, less than 50k or greater). Now, we start the process. The aim of the process is to maximum correct answers by building 20 teams having 5 members each. Any member can be a part of more than 1 team, and any member can have more than 1 more than 1 role in same team. The member which have maximum roles is the most important variable of our model.

The process starts with a random guess of answers. Then it calculates  $error = Actual - Predicted Answer$ . Next step, it build a team of 5 members, which reduces the error by maximum. Again, it calculates the error. The second team (tree) has to reduce it further. But next team doesn't trust its previous partner fully, so it assume that answers are correct with x probability (learning

rate). This process goes on till 20 teams are built. So in the process, we have to decide, how many teams to build (trees), members in each team (depth) and learning team, so that error in the end is minimum. This can only be done by trial and error method.

---

A gradient boosting model is a **CHAIN** of decision trees that also each make a vote. But instead of each learning in isolation, when you add a new one to the chain, it tries to improve a bit on what the rest of the chain already thinks. So, a new tree's decision is influenced by all the trees that have already voiced an opinion. When you add a new tree to a GBC, it gets to see what its predecessors thought - and how they got it right or wrong. They then formulate a suggestion to correct the errors of their predecessors - and then they add that to the pot, and then the process continues with the next tree you add to the chain.

---

It requires three things, a loss/error function, a weak learning model, and an additive model. First we start by finding the loss or error function for the problem. In our case the loss function is logarithmic (since we are dealing with classification problem). Next, we use decision trees to create numerous 'weak' learning models. 'Weak learners' are models that are slightly better than chance, meaning that the model will predict correctly slightly more than 50% of the time.

Now that we have a large number of decision trees that are simple to build and run, we move onto our additive model function. The GBC model looks at each individual decision tree created and will add together the trees to create a single modeled tree with the lowest possible loss/error. Every node, every branch, every level is analyzed and the additive model will help to create the most efficient model possible.

---

**Resources:** - [Quora](#) - [Machine Learning Mastery](#)

### 1.5.3 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 [18]: # TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
         from sklearn.grid_search import GridSearchCV
```



```

from sklearn.metrics import make_scorer

# TODO: Initialize the classifier
clf = GradientBoostingClassifier(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 = {'n_estimators': [100, 300, 500]}

# 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 using GridSearchCV
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-after scores
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: 0.8630

F-score on testing data: 0.7395

Optimized Model

-----

Final accuracy score on the testing data: 0.8718

Final F-score on the testing data: 0.7545

In [19]: print(best\_clf)

```

GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,

```

```
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=500,
presort='auto', random_state=0, subsample=1.0, verbose=0,
warm_start=False)
```

#### 1.5.4 Question 5 - Final Model Evaluation

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

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

Metric	Unoptimized Model	Optimized Model
Accuracy Score	0.8638	0.8718
F-score	0.7395	0.7545

**Results: Answer:** - The accuracy score and f\_score of an Optimized Model are quite better than an Unoptimized model. We see more than a percent increase in accuracy and 0.015 percent increase in F\_score, its seems not quite that much but it helps the model alot and will obviously better predict than an Unoptimized Model.

- Comparing this answer with above naive predictor's benchmark in **Q1**, this model is 63% better in terms of accuracy and 36% better in terms of F\_score compared to Naive Predictor.

## 1.6 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.

### 1.6.1 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:** - **Capital-loss:** It is critical to determine who might be willing to donate if they are facing loss. The more loss they are in, the more less likely they will be donating. How much wealthy they are doesn't matter, when someone is losing assets or having huge losses, it's difficult to say that they will donate. - **Capital-gain:** The opposite of capital-loss. An individual making more money has a higher probability that they will donate more. Even an individual who doesn't donate might be willing to donate as after all that individual is having a good time and life with it. - **Occupation:** The higher your rank or position in Industry, the higher you make. - **Education level:** Same argument I would make here as I did for occupation but even higher level of education doesn't guarantee you that you are going to earn that much as compared to occupation. - **Age:** Age plays an important role since people who are mid-aged have a high probability of earning more and have less probability for a senior level aged.

### 1.6.2 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_'`.

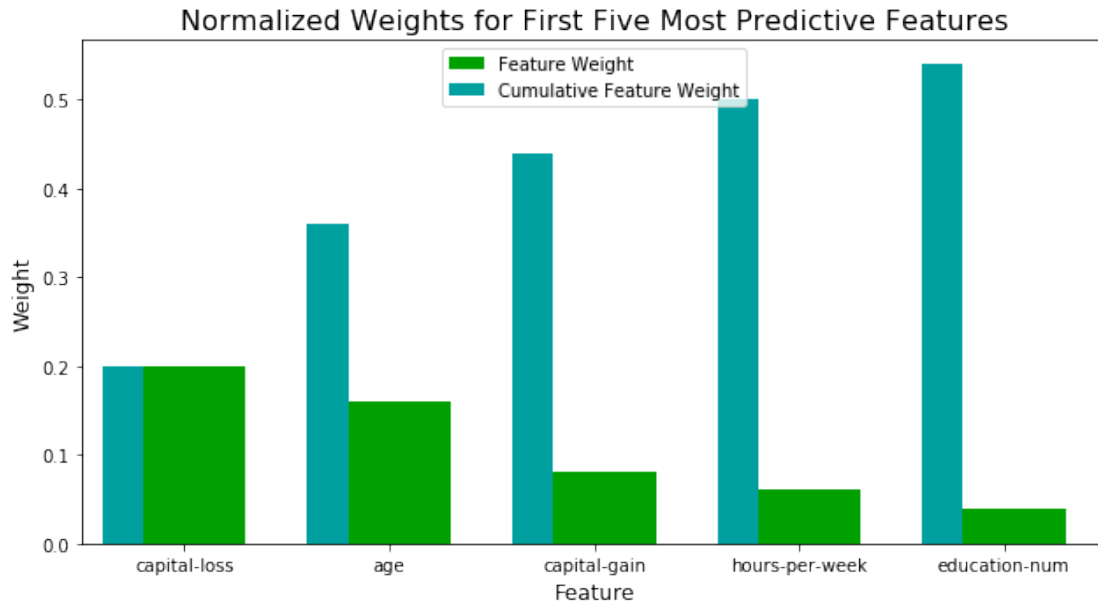
```
In [24]: # TODO: Import a supervised learning model that has 'feature_importances_'
         from sklearn.ensemble import AdaBoostClassifier

         model = AdaBoostClassifier(random_state=0)

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



### 1.6.3 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:** I am quite surprised that I got almost all right except for one which is 'hours-per-week'. I didn't expect this to be on top 5 and I am also stunned that 'Occupation' is nowhere to be seen in top 5 since I had made bold confident statements above. And it also seems age is a crucial factor here, though I had mentioned it in my top 5, I didn't expect to be so high based on my intuition. Numbers and data don't lie, 'age' having more influence than 'capital-gain' is quite surprising.

### 1.6.4 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 [25]: # Import functionality for cloning a model
         from sklearn.base import clone
```

```

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

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

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

# Report scores from the final model using both versions of data
print("Final Model trained on full data\n-----")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, best_predictions)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, 1)))
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: 0.8718

F-score on testing data: 0.7545

Final Model trained on reduced data

-----

Accuracy on testing data: 0.8425

F-score on testing data: 0.6993

### 1.6.5 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:** - As we can see, reduced data gives us reduced accuracy and F\_score. It's about 3 percent decrease in accuracy and about 5 percent in F-score. - Feeding full census data to Optimized GDC takes obviously more time than feeding reduced census data. It took about 5-10 min to model full census data and it took seconds for reduced data. - Now, When it comes to accuracy and F-score, we are trading off crucial scores when we feed less data which takes very less time to model but when we feed full data we do end up getting better results with slightly higher time rate. Its a simple trade off we make here, if we want it quick, we would feed reduced data, hence ending up with lower accuracy and fscore and the opposite argument I would make for full data. - I personally would stick to feeding full data. Since it doesn't take much longer time to model full data than reduced data, the important thing we get here is better scores. Once modeled, classifier can classify results instant. So a one time trade-off of time , gives us a lifetime of better results.