

Introduction to Machine Learning with PyTorch

Project: Finding Donors for *CharityML*

Welcome to the first project of the Data Scientist 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](#). 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.

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.

Python 3.10.7 is being used for this project

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	capital-loss
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	0

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]:

```
data.income.value_counts()
```

Out[2]:

```
income
<=50K    34014
>50K     11208
Name: count, dtype: int64
```

In [3]:

```
n_records = data.shape[0]

n_greater_50k = data.query('income == ">50K"').shape[0]

n_at_most_50k = data.query('income == "<=50K"').shape[0]
```

```

greater_percent = round(n_greater_50k/n_records * 100, 2)

# 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: 24.78%

```

Featureset Exploration

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

Preparing the Data

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

Transforming Skewed Continuous Features

A dataset may sometimes contain at least one feature whose values tend to lie near a single number, but will

also have a non-trivial number of vastly larger or smaller values than that single number. Algorithms can be sensitive to such distributions of values and can underperform if the range is not properly normalized. With the census dataset two features fit this description: `capital-gain` and `capital-loss`.

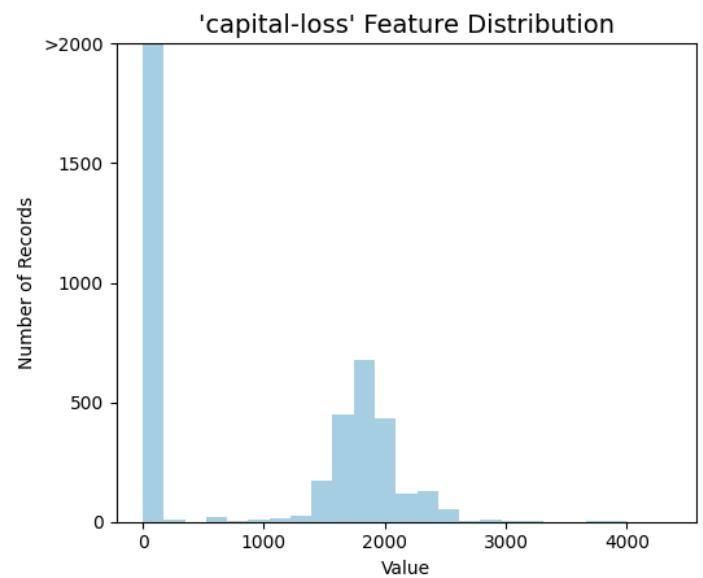
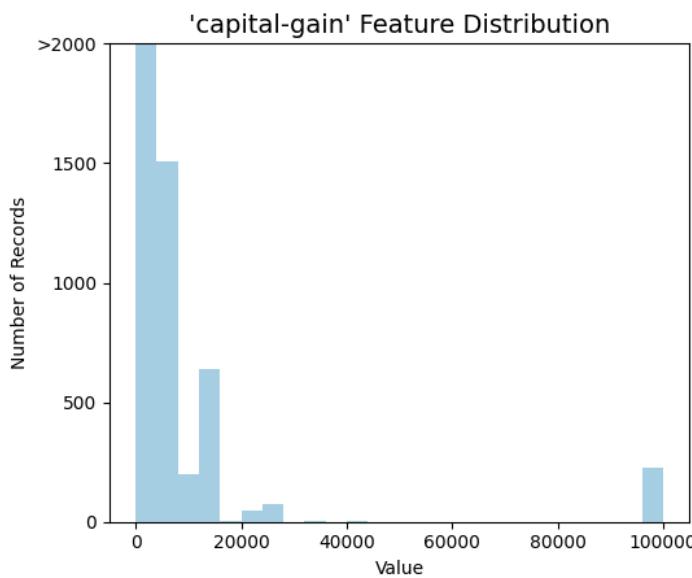
Run the code cell below to plot a histogram of these two features. Note the range of the values present and how they are distributed.

```
In [4]: # Split the data into features and target label
```

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

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

Skewed Distributions of Continuous Census Data Features



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

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

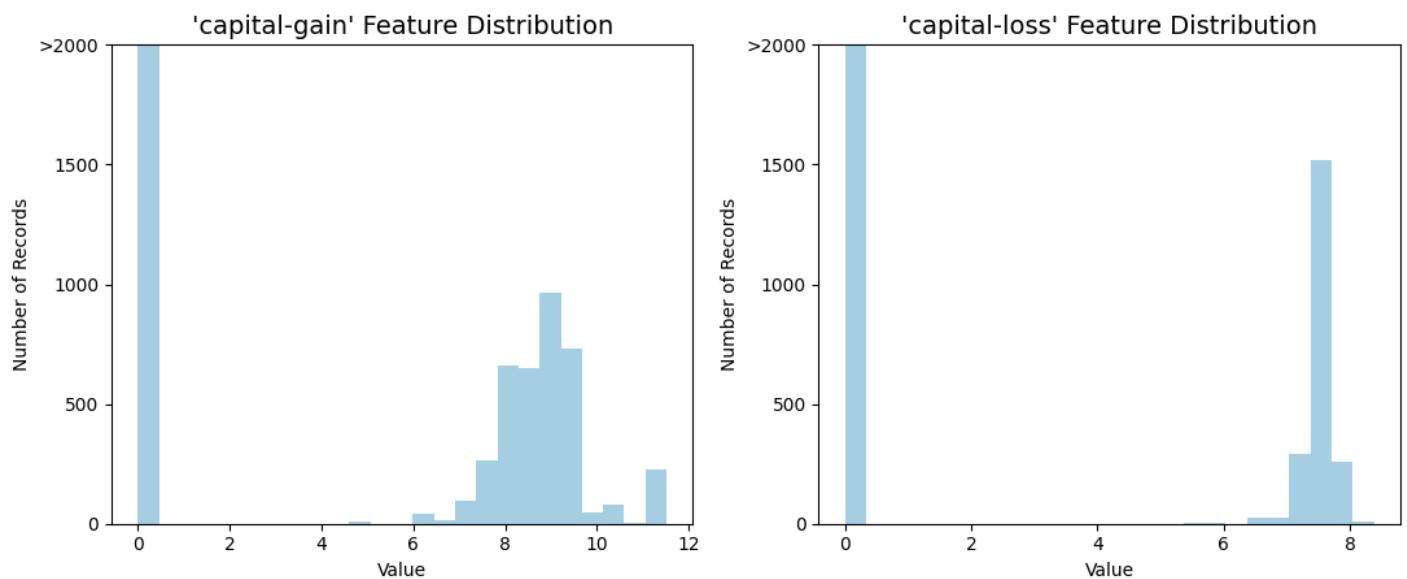
```
In [5]: # Log-transform the skewed features
```

```
skewed = ['capital-gain', 'capital-loss']
features_log_transformed = pd.DataFrame(data = features_raw)
features_log_transformed[skewed] = features_raw[skewed].apply(lambda x: np.log(x + 1))
```

```
# Visualize the new Log distributions
```

```
vs.distribution(features_log_transformed, transformed = True)
```

Log-transformed Distributions of Continuous Census Data Features



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

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

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

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

	age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	cap
0	0.301370	State-gov	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family	White	Male	0.66
1	0.452055	Self-emp-not-inc	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.00
2	0.287671	Private	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.00
3	0.493151	Private	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.00
4	0.150685	Private	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.00

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 :

someFeature	
0	B
1	C
2	A

We then encode this feature into `someFeature_A` , `someFeature_B` and `someFeature_C` :

	someFeature_A	someFeature_B	someFeature_C
0	0	1	0
1	0	0	1
2	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 " $\leq 50K$ " to 0 and records with " $> 50K$ " to 1

```
In [7]: features_final = pd.get_dummies(data=features_log_minmax_transform, dtype=int)

income = income_raw.apply(lambda x: 0 if x == "<=50K" else 1)

# 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-government', 'workclass_Local-gov', 'workclass_Private', 'workclass_Self-emp-inc', 'workclass_Self-employed-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_Own-child', 'relationship_Unmarried', 'relationship_Wife', 'race_Amer-Indian-Eskimo', 'race_Asian-Pac-Islander', 'race_Black', 'race_Other', 'race_White', 'sex_Female', 'sex_Male', 'native-country_Cambodia', 'native-country_Canada', 'native-country_China', 'native-country_Columbia', 'native-country_Cuba', 'native-country_Dominican-Republic', 'native-country_Ecuador', 'native-country_El-Salvador', 'native-country_England', 'native-country_France', 'native-country_Germany', 'native-country_Greece', 'native-country_Guatemala', 'native-country_Haiti', 'native-country_Holand-Netherlands', 'native-country_Honduras', 'native-country_Hong', 'native-country_Hungary', 'native-country_India', 'native-country_Iran', 'native-country_Ireland', 'native-country_Italy', 'native-country_Jamaica', 'native-country_Japan', 'native-country_Laos', 'native-country_Mexico', 'native-country_Nicaragua', 'native-country_Outlying-US(Guam-USVI-etc)', 'native-country_Peru', 'native-country_Philippines', 'native-country_Poland', 'native-country_Portugal', 'native-country_Puerto-Rico', 'native-country_Scotland', 'native-country_South', 'native-country_Taiwan', 'native-country_Thailand', 'native-country_Trinidad&Tobago', 'native-country_United-States', 'native-country_Vietnam', 'native-country_Yugoslavia']

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.

```
test_size = 0.2,  
random_state = 0)
```

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

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

Evaluating Model Performance

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

Metrics and the Naive Predictor

CharityML, equipped with their research, knows individuals that make more than 50,000 are most likely to donate to their charity. Because of this, *CharityML* is particularly interested 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 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 classification), in other words it is the ratio of

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

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

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

For classification problems that are skewed in their classification distributions, like in our case, for example, if we had 100 text messages and only 2 were spam and the remaining 98 weren't, accuracy 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 and the 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 when 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.

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

HINT:

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

In [9]: `data.head()`

Out[9]:

	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	2174.0
1	50	Self-emp-not-inc	Bachelors	13.0	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.0
2	38	Private	HS-grad	9.0	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.0
3	53	Private	11th	7.0	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.0
4	28	Private	Bachelors	13.0	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.0

In [10]:

```
TP = np.sum(income) # Counting the ones as this is the naive case. Note that 'income' is the 'income' column from the dataset
FP = income.count() - TP # Specific to the naive case

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

accuracy = TP/income.count()
recall = TP/(TP+FP)
precision = TP/(TP+FN)

B = 0.5
fscore = (1 + B**2) * ((precision * recall)/((B**2 * precision) + recall))

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

Naive Predictor: [Accuracy score: 0.2478, F-score: 0.6223]

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

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: GaussianNB

- Real-world application: Spam detection by analyzing common words, length, etc.
- Strengths/when does it work well:
 - Small datasets
 - Continuous data
 - Fast training and prediction
- Weaknesses/when does it work poorly:
 - Naive assumption can hurt real-world accuracy
 - Not the best for complex decision boundaries
- Why is this model a good candidate:
 - If the distributions are bell-shaped
 - Features vary independently

Decision Trees

- Real-world application:
 - Emergency room triage - assessing the severity of chest pain, what symptoms are experienced, and who would be the best medical professional to take care of the patient (sourced from: <https://mljourney.com/decision-tree-real-life-examples/>)
- Strengths/when does it work well:
 - Easy to visualize/explain the model
 - Requires very little preprocessing
- Weaknesses/when does it work poorly:
 - Prone to overfitting
 - Poor performance with small datasets (without regularization)
- Why is this model a good candidate:
 - Contains categorical and numerical features, which can be handled without scaling
 - Clear and interpretable results

Logistic Regression

- Real-world application: binary classifications. Detecting if logins and certain network activity is anomalous. (<https://www.sciencedirect.com/science/article/abs/pii/S1084804519303200>)
- Strengths/when does it work well:
 - Easy to train

- Handles noise (when regularized)
- Works great for binary classification (true or false, yes or no, will donate or won't, etc.)
- Weaknesses/when does it work poorly:
 - Requires adjustments with imbalanced classes
- Why is this model a good candidate:
 - If the relationship between the features and the person's income are generally linear
 - Highly interpretable

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 [11]: 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 = {}

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

    results['train_time'] = round(end - start, 2)

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

```

results['pred_time'] = round(end - start, 2)

results['acc_train'] = accuracy_score(y_train[:300], predictions_train)

results['acc_test'] = accuracy_score(y_test, predictions_test)

results['f_train'] = fbeta_score(y_train[:300], predictions_train, beta=B)

results['f_test'] = fbeta_score(y_test, predictions_test, beta=B)

# 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]: from sklearn.tree import DecisionTreeClassifier
         from sklearn.naive_bayes import GaussianNB
         from sklearn.linear_model import LogisticRegression

clf_A = DecisionTreeClassifier()
clf_B = GaussianNB()
clf_C = LogisticRegression()

# HINT: samples_100 is the entire training set i.e. len(y_train)
# HINT: samples_10 is 10% of samples_100 (ensure to set the count of the values to be `int` and not
# HINT: samples_1 is 1% of samples_100 (ensure to set the count of the values to be `int` and not
samples_100 = y_train.shape[0]
samples_10 = samples_100 // 10
samples_1 = samples_100 // 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)

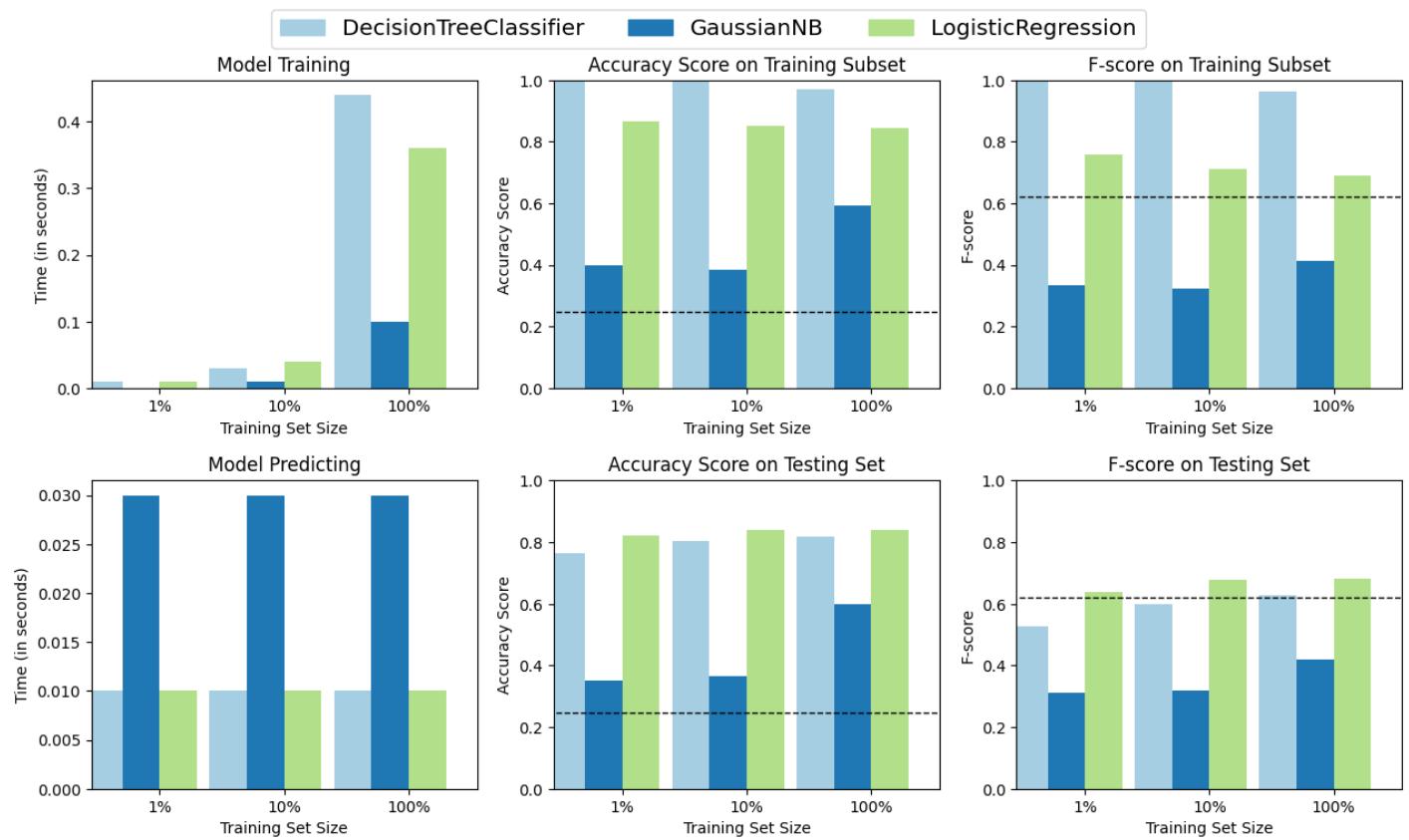
# Run metrics visualization for the three supervised Learning models chosen
vs.evaluate(results, accuracy, fscore)

DecisionTreeClassifier trained on 361 samples.
DecisionTreeClassifier trained on 3617 samples.
DecisionTreeClassifier trained on 36177 samples.
GaussianNB trained on 361 samples.
GaussianNB trained on 3617 samples.
GaussianNB trained on 36177 samples.
LogisticRegression trained on 361 samples.
LogisticRegression trained on 3617 samples.
LogisticRegression trained on 36177 samples.

c:\Users\Ethan\OneDrive\Documents\Code\Python Projects\machine-learning\sup-ml-project\visuals.py:121: UserWarning: Tight layout not applied. tight_layout cannot make Axes width small enough to accommodate all Axes decorations
    pl.tight_layout()

```

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: Based on evaluation above, the most appropriate of the 3 models tested above would be Logistic Regression. Not only does the model train and predict quickly, but it also doesn't show signs of overfitting the training data based on the somewhat consistent and still solid F-Scores against the training data and the superior F-Scores on the testing set. It is also significantly better on F-Score and Accuracy when compared to GaussianNB .

In terms of suitability for the data, as well as the model's purpose, logistic regression is perfect for this. Not only is it highly interpretable, meaning you can see how much each of the features effect the predictions, but it's among the best for predicting binary outcomes. *CharityML*'s goal here is to determine who will probably donate versus who will probably not.

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.

Answer:

Logistic regression is used to predict the probability of how likely something or someone falls into one of two buckets. For example, it can be used against financial transactions to determine if they are fraudulent or not. It uses each of the variables passed to it, like in your use case, a person's age or income level, and assigns a "coefficient" to it. This is essentially a set of dials that the algorithm tunes during training and is then used to draw an S-Shaped curve on a graph. Then, you feed your testing data, which could be plotted like points on a graph. That curve is then "overlaid" on the graph and each point is put into either the "might" (makes $\geq 50K$) or "won't" donate bucket, depending on where on the graph they sit in regards to the curve. If they are above the curve, they are more likely to have an income over \$50K. Lower means that they likely make less.

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 [ ]: from sklearn.metrics import make_scorer
from sklearn.model_selection import GridSearchCV

clf = LogisticRegression(max_iter=1000)

# HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, value2]}
parameters = {
    'C': [.01, 0.1, 1, 10],
    'class_weight': [None, 'balanced'],
}

scorer = make_scorer(fbeta_score, beta=B)

grid_obj = GridSearchCV(clf, param_grid=parameters, scoring=scorer, n_jobs=-1)

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)))

In [ ]: print(grid_obj.best_params_)

{'C': 0.1, 'class_weight': None}
```

Question 5 - Final Model Evaluation

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

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

Results:

Metric	Unoptimized Model	Optimized Model
Accuracy Score	84.17%	84.21%
F-score	0.683	0.684

Answer:

The model was tuned by adjusting the `C` value to be `0.1` instead of the default of `1.0`. The other parameter we were testing, being the class weight, will be left as `None`, which is the default. The accuracy score and F-score for the optimized model are only very mildly better than the unoptimized model. In comparison to the naive predictor benchmarks from question 1 ([Accuracy score: `0.2478`, F-score: `0.6223`]), the optimized model's accuracy is significantly better and the F-score has a fair 6.2% improvement.

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:

Top 5 most important for prediction

1. **Capital-gain** - Wealth beyond income earned directly from work indicates that a person can afford to start a passive income stream, and therefore makes much more than \$50K/year.
2. **Marital status** - With more possibly working adults in a family, it's more likely that they will have a total income at or below \$50K.
3. **Age** - May be a decent indicator for financial stability, which also means someone would be more able to donate because they make over \$50K. Older individuals have likely been in the workforce longer, and

have therefore promoted to more senior positions (at least, not entry-level).

4. **Occupation** - White-collar professions are more likely to be paid salaries, which are often for full time positions that make near or more than \$50K.
5. **Working class** - One's sector of work can indicate how certain it is that they make over 50K. For example, a large portion of federal government positions (GS pay scale) are GS – 9 or higher 51,000 in base pay.

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:

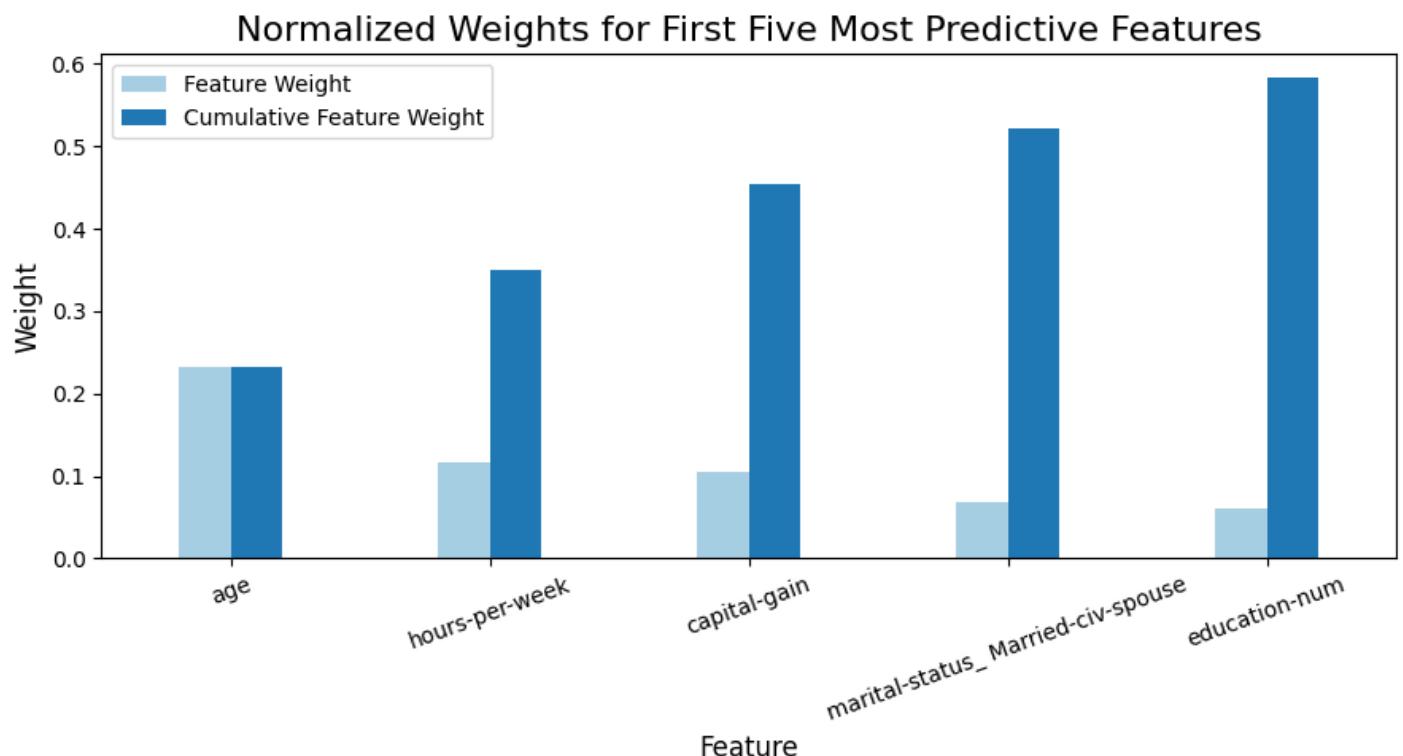
- Import a supervised learning model from `sklearn` if it is different from the three used earlier.
- Train the supervised model on the entire training set.
- Extract the feature importances using `.feature_importances_`.

```
In [ ]: from sklearn.ensemble import RandomForestClassifier
```

```
model = RandomForestClassifier().fit(X_train, y_train)

importances = model.feature_importances_

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



Question 7 - Extracting Feature Importance

Observe the visualization created above which displays the five most relevant features for predicting if an individual makes at most or above \$50,000.

- How do these five features compare to the five features you discussed in **Question 6**?
- If you were close to the same answer, how does this visualization confirm your thoughts?
- If you were not close, why do you think these features are more relevant?

Answer:

I wasn't really close at all.

1. **Incorrect:** I guessed capital gain while it was actually age. This makes sense, as for the reason I said age was at least in the third spot - age indicates how long someone has been working, and it's likely that (generally) people who have worked for longer fill more senior positions.
2. **Incorrect:** Hours per week instead of marital status. This makes some sense as the number of hours worked per week probably is a good indicator of income, but also probably coincides with age.
3. **Incorrect:** I put age here instead of capital gain. It would make sense that age would be a "heavier" feature than capital gain, as it's more likely someone who is older, but without much or any capital gain, is still somewhat likely to make over 50K. *Capital gain is probably a better indicator for someone making much more than 50K, since there is a high startup cost to most passive revenue streams.*
4. **Incorrect:** I had occupation here instead of someone being married to a civilian spouse. It makes sense that this would be on here instead of occupation, as having twice the amount of working adults in a family means that one family may make generally more than a single person.
5. **Incorrect:** I had working class here instead of education level. It makes sense that education level means one is more likely to be paid more than \$50,000, and it probably even is an indicator for what occupational sector someone likely resides in.

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 [ ]: # 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, beta = 0.5)))
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, beta = 0.5)))
```

```
Final Model trained on full data
-----
Accuracy on testing data: 0.8421
F-score on testing data: 0.6843

Final Model trained on reduced data
-----
Accuracy on testing data: 0.8263
F-score on testing data: 0.6477
```

Question 8 - Effects of Feature Selection

- How does the final model's F-score and accuracy score on the reduced data using only five features compare to those same scores when all features are used?
- If training time was a factor, would you consider using the reduced data as your training set?

Answer:

When compared to our optimized Logistic Regression model, using only the top 5 features led to a ~3.7 point reduction in F-Score and a ~1.6% reduction in accuracy. This means that the recall and precision of the model suffered, but not to a severe degree. Essentially, by using about a third of the number of features, we lost a meager amount of performance. Additionally, we still cover 60% of the total weights for the model.

This reduction seems like it could be worth it if training time is a factor, especially since the loss on accuracy is so minimal. I would recommend including a few more of the top features, perhaps the top 7, in order to minimize the loss in accuracy, recall, and precision even further.

```
In [ ]: !jupyter nbconvert --to html finding_donors.ipynb
```

```
[NbConvertApp] Converting notebook finding_donors.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 4 image(s).
[NbConvertApp] Writing 625814 bytes to finding_donors.html
```