finding_donors

July 9, 2022

0.1 Supervised Learning

0.2 Project: Finding Donors for *CharityML*

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.

0.3 Getting Started

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

The dataset for this project originates from the UCI Machine Learning Repository. The datset was donated by Ron Kohavi and Barry Becker, after being published in the article *"Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid"*. You can find the article by Ron Kohavi online. The data we investigate here consists of small changes to the original dataset, such as removing the `'fnlwgt'` feature and records with missing or ill-formatted entries.

0.4 Exploring the Data

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

```
In [2]: # Import libraries necessary for this project
        import numpy as np
        import pandas as pd
        from time import time
        from IPython.display import display # Allows the use of display() for DataFrames

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

        # Pretty display for notebooks
        %matplotlib inline

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

        # Success - Display the first record
        display(data.head(n=10))
```

	age	workclass	education_level	education-num \
0	39	State-gov	Bachelors	13.0
1	50	Self-emp-not-inc	Bachelors	13.0
2	38	Private	HS-grad	9.0
3	53	Private	11th	7.0
4	28	Private	Bachelors	13.0
5	37	Private	Masters	14.0
6	49	Private	9th	5.0
7	52	Self-emp-not-inc	HS-grad	9.0
8	31	Private	Masters	14.0
9	42	Private	Bachelors	13.0

	marital-status	occupation	relationship	race \
0	Never-married	Adm-clerical	Not-in-family	White
1	Married-civ-spouse	Exec-managerial	Husband	White
2	Divorced	Handlers-cleaners	Not-in-family	White
3	Married-civ-spouse	Handlers-cleaners	Husband	Black
4	Married-civ-spouse	Prof-specialty	Wife	Black
5	Married-civ-spouse	Exec-managerial	Wife	White
6	Married-spouse-absent	Other-service	Not-in-family	Black
7	Married-civ-spouse	Exec-managerial	Husband	White

```
8              Never-married      Prof-specialty   Not-in-family   White
9         Married-civ-spouse      Exec-managerial        Husband   White

          sex   capital-gain   capital-loss   hours-per-week   native-country   income
0        Male         2174.0            0.0             40.0    United-States    <=50K
1        Male            0.0            0.0             13.0    United-States    <=50K
2        Male            0.0            0.0             40.0    United-States    <=50K
3        Male            0.0            0.0             40.0    United-States    <=50K
4      Female            0.0            0.0             40.0             Cuba    <=50K
5      Female            0.0            0.0             40.0    United-States    <=50K
6      Female            0.0            0.0             16.0          Jamaica    <=50K
7        Male            0.0            0.0             45.0    United-States     >50K
8      Female        14084.0            0.0             50.0    United-States     >50K
9        Male         5178.0            0.0             40.0    United-States     >50K
```

0.4.1 Implementation: Data Exploration

A cursory investigation of the dataset will determine how many individuals fit into either group, and will tell us about the percentage of these individuals making more than $50,000. In the code cell below, you will need to compute the following: - The total number of records, `'n_records'` - The number of individuals making more than $50,000 annually, `'n_greater_50k'`. - The number of individuals making at most $50,000 annually, `'n_at_most_50k'`. - The percentage of individuals making more than $50,000 annually, `'greater_percent'`.

** HINT: ** You may need to look at the table above to understand how the `'income'` entries are formatted.

```python
In [3]: # TODO: Total number of records
        n_records = None
        n_records = len(data.index)
        # TODO: Number of records where individual's income is more than $50,000
        n_greater_50k = None
        n_greater_50k = data['income'].value_counts()['>50K']
        # TODO: Number of records where individual's income is at most $50,000
        n_at_most_50k = None
        n_at_most_50k = data['income'].value_counts()['<=50K']
        # TODO: Percentage of individuals whose income is more than $50,000
        greater_percent = None
        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, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

0.5   Preparing the Data

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

0.5.1   Transforming Skewed Continuous Features

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

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

```
In [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
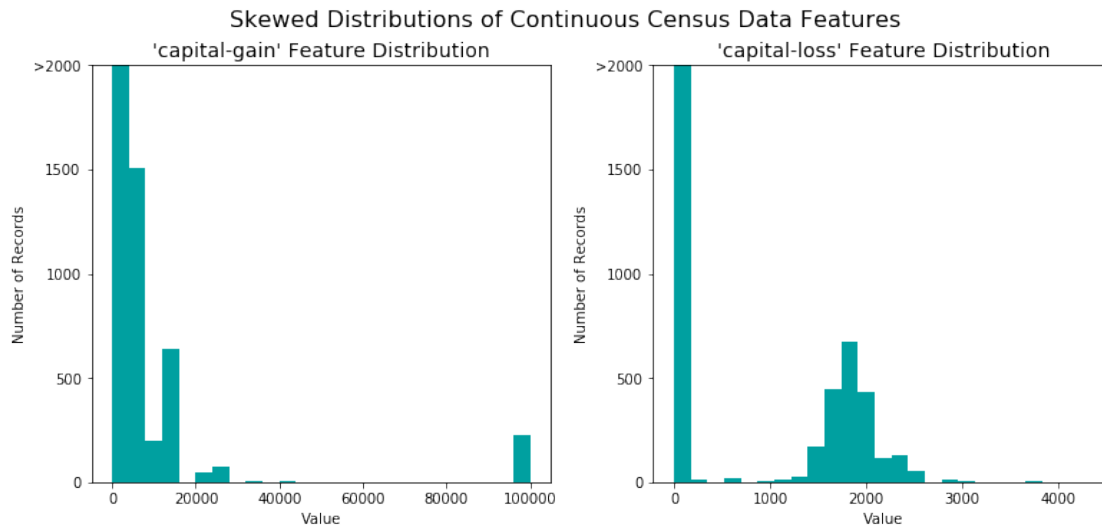
For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'`, it is common practice to apply a logarithmic transformation on the data so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation 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
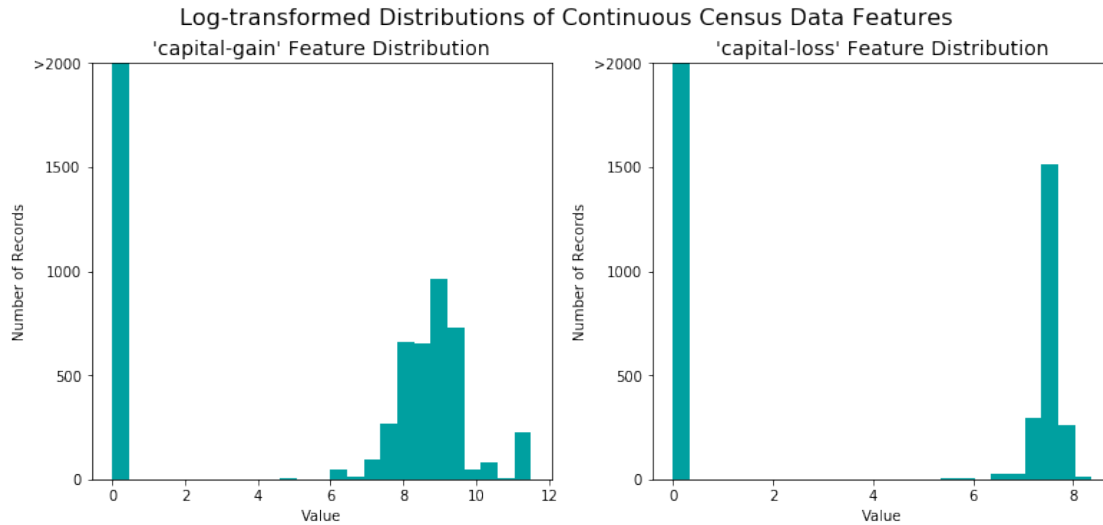
0.5.2 Normalizing Numerical Features

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

Run the code cell below to normalize each numerical feature. We will use `sklearn.preprocessing.MinMaxScaler` for this.

```
In [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

         # 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
```

6

```
         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
```

0.5.3   Implementation: Data Preprocessing

From the table in **Exploring the Data** above, we can see there are several features for each record
that are non-numeric. Typically, learning algorithms expect input to be numeric, which requires
that non-numeric features (called *categorical variables*) be converted. One popular way to con-
vert 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.

```python
In [7]:  # TODO: One-hot encode the 'features_log_minmax_transform' data using pandas.get_dummies
         features_final = None
         categorical_cols = ['workclass', 'education_level', 'marital-status', 'occupation', 'rel
         for col in categorical_cols:
             tempdf = pd.get_dummies(features_log_minmax_transform[col], prefix = col)
             features_log_minmax_transform = pd.merge(
                 left=features_log_minmax_transform,
                 right=tempdf,
                 left_index=True,
                 right_index=True,
```

7

```
    )
        features_log_minmax_transform = features_log_minmax_transform.drop(columns=col)
    features_final = features_log_minmax_transform
    # TODO: Encode the 'income_raw' data to numerical values
    income = None
    income = income_raw.map({'>50K': 1 , '<=50K' : 0})
    # Print the number of features after one-hot encoding
    encoded = list(features_final.columns)
    print("{} total features after one-hot encoding.".format(len(encoded)))

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

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

0.5.4 Shuffle and Split Data

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

Run the code cell below to perform this split.

```
In [8]: # Import train_test_split
        from sklearn.model_selection 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.
```

Note: this Workspace is running on $sklearn$ v0.19. If you use the newer version (>="0.20"), the $sklearn.cross\_validation$ has been replaced with $sklearn.model\_selection$.

0.6 Evaluating Model Performance

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

8

0.6.1 Metrics and the Naive Predictor

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

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

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

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

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).

0.6.2   Question 1 - Naive Predictor Performace

- If we chose a model that always predicted an individual made more than $50,000, what would that model's accuracy and F-score be on this dataset? 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 [9]:  '''
         TP = np.sum(income) # Counting the ones as this is the naive case. Note that 'income' is
         encoded to numerical values done in the data preprocessing step.
         FP = income.count() - TP # Specific to the naive case

         TN = 0 # No predicted negatives in the naive case
         FN = 0 # No predicted negatives in the naive case
         '''
         # TODO: Calculate accuracy, precision and recall
         accuracy = None
         recall = None
         precision = None

         TP = np.sum(income)
         precision = TP/n_records
         accuracy = precision
         recall = 1

         # TODO: Calculate F-score using the formula above for beta = 0.5 and correct values for
         fscore = None
         fscore = (1 + (0.5)**2) * ((precision * recall)/((0.5**2 * precision) + recall))
         # Print the results
         print("Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".format(accuracy, fsco

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

0.6.3 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

0.6.4 Question 2 - Model Application

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

- Describe one real-world application in industry where the model can be applied.
- 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 aboveˆ, with 4 parts for each of the three models you pick. Please include references with your answer.

Answer:
Used These Websites: https://elitedatascience.com/machine-learning-algorithms#:~:text=Strengths%3A%20Outputs%20have%20a%20nice,or%20non%2Dlinear%20decision%20boun...
https://activewizards.com/blog/5-real-world-examples-of-logistic-regression-application

AdaBoost: 1. Is a classification algorithm, can be applied to predicting whether or not a patient has a certain disease based on previous health history, any problem in which a yes/no decision can be made 2. This model combines three different models into one to minimize error. The first model is a decision tree based off of the training data, the second model builds off of the first to increase accuracy, and the third again aims to reduce error. This boosting technique, where one model aims to better predict than the next by changing the weights on correct and incorrect predictions, helps to reduce bias error. 3. If the dataset has a lot of outliers, then it will not make as accurate a prediction because the model will not know what values to build off of. 4. This model makes a good candidate for the problem because most input values are categorical, which are now binary, so there cannot be that many outliers in the dataset. Additionally, if the model places initial weights on the inputs, these weights can be changed depending on whether the prediction for target income was correct or not pretty easily, since there are only two values for most of the variables and the numerical columns have been transformed to lie between 0 and 1. Additionally, because the question is ultimately asking the model to make a decision about a certain row of data, a sequential algorithm would be useful to make this decision because it depends on the errors of its predecessors and can therefore improve its prediction.

Support Vector Machines 1. Can be used to classify voters as Republican or Democrat based on their personal data 2. This algorithm performs well when there is a clean cut between 2 sections of the data. It also performs better when there are higher dimension to the hyperplane that separates the two data sections since more dimensions means more flexibility 3. This algorithm does not perform well when the dataset is large, because this means that the hyperplane will have to readjust several times for each point in the set and there are more points. It will also not do well when the number of features in a dataset is larger than the number of values in the training

11

data because this means that the number of dimensions is greater than the number of values, so the model will not be able to create the hyperplane. 4. This model makes a good candidate for the problem because there is enough data for the model to create a boundary and margins for 2 separate sections but not too much where it will take too much time. Additionally, while this problem does not have a clear distinction in its data just by looking at the table, it does have several categorical values that have been converted into binary values that might be the same for each entry depending on if the target value is <=50K or >50K.

Logistic Regression 1. Is used in predictive models for credit scoring 2. This algorithm's strengths include that it is easily interpretable, it's easy to find out which inputs have a larger impact on the final result of the prediction, and the optimal number of features can be found . 3. It cannot necessarily capture more complex relationships or multiple features and decision boundaries. 4. Logistic Regression is easy to interpret, there is one target variable so logistic regression will be able to classfiy a data point in one of 2 ways.

0.6.5 Implementation - Creating a Training and Predicting Pipeline

To properly evaluate the performance of each model you've chosen, it's important that you create a training and predicting pipeline that allows you to quickly and effectively train models using various sizes of training data and perform predictions on the testing data. Your implementation here will be used in the following section. In the code block below, you will need to implement the following: - Import `fbeta_score` and `accuracy_score` from `sklearn.metrics`. - Fit the learner to the sampled training data and record the training time. - Perform predictions on the test data `X_test`, and also on the first 300 training points `X_train[:300]`. - Record the total prediction time. - Calculate the accuracy score for both the training subset and testing set. - Calculate the F-score for both the training subset and testing set. - Make sure that you set the `beta` parameter!

```python
In [10]: # TODO: Import two metrics from sklearn - fbeta_score and accuracy_score
         from sklearn.metrics import fbeta_score
         from sklearn.metrics import 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.fit(X_train, y_train)
             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 .pred
            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
```

0.6.6   Implementation: Initial Model Evaluation

In the code cell, you will need to implement the following: - Import the three supervised learning models you've discussed in the previous section. - Initialize the three models and store them in `'clf_A'`, `'clf_B'`, and `'clf_C'`. - Use a `'random_state'` for each model you use, if provided. - **Note:** Use the default settings for each model — you will tune one specific model in a later section. - Calculate the number of records equal to 1%, 10%, and 100% of the training data. - Store those values in `'samples_1'`, `'samples_10'`, and `'samples_100'` respectively.

   Note: Depending on which algorithms you chose, the following implementation may take some time to run!

```
In [11]:  # TODO: Import the three supervised learning models from sklearn
          from sklearn.linear_model import LogisticRegression
          from sklearn.ensemble import AdaBoostClassifier
          from sklearn.svm import SVC

          # TODO: Initialize the three models
          clf_A = LogisticRegression(random_state = 42)
```

```python
            clf_B = AdaBoostClassifier(random_state = 42)
            clf_C = SVC(random_state = 42)

            # 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 `in
            samples_100 = len(y_train)
            samples_10 = int(samples_100/10)
            samples_1 = int(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)

LogisticRegression trained on 361 samples.
LogisticRegression trained on 3617 samples.
LogisticRegression trained on 36177 samples.
AdaBoostClassifier trained on 361 samples.
AdaBoostClassifier trained on 3617 samples.
AdaBoostClassifier trained on 36177 samples.
SVC trained on 361 samples.
SVC trained on 3617 samples.
SVC trained on 36177 samples.
```
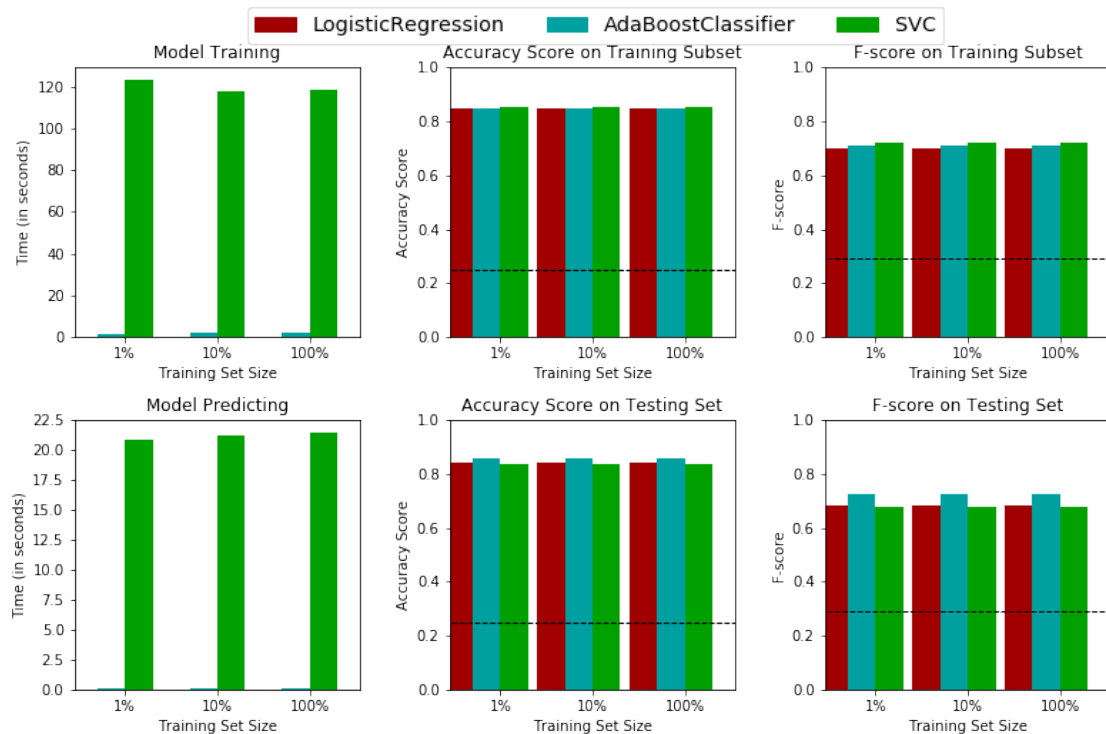
Performance Metrics for Three Supervised Learning Models

0.7 Improving Results

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

0.7.1 Question 3 - Choosing the Best Model

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

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

Answer:

The AdaBoostClassifier model, which is an ensemble supervised learning method, is the most appropriate for the task of identifying individuals that make more than $50,000. This is because

15

this model has the greatest f-score on 100 percent of the testing set, takes a very short amount of time to both train and predict, and can be applied to any number of values in the set. When looking at the metrics, Logisitc Regression, AdaBoost, and SVC were all close in performance, with SVC being sligthly higher in accuracy and f-score on the training set, but on the testing set, Adaboost was higher in both accuracy and f-score, which indicates that AdaBoost is a better predictor of whether or not a certain individual makes above or below 50K. Comparing the sizes of the sets used (ie when 1, 10, or 100 percent) of the training data is used, the answer remains the same because the differences in scores for each of the respective models is negligible across the training set sizes. In terms of prediction time, Logistic Regression took the least amount of time, as you can not even see it on the charts comparing Model Training and Model Predicting time. However, AdaBoost does not take much time either, as you can barely see it on the Model Predicting chart, indicating that there is not much of a time difference between the two and therefore time does not have as much of an impact when comparing performance between the two. However, SVC takes a significantly longer amount of time to perform, which means that SVC is not as efficient of an algorithm to use, probably because the dataset is very large. This means that although accuracy and f-score between AdaBoost and SVC are close, SVC is not the best algorithm to use. AdaBoost has the highest f-score, meaning that precision for the model is higher than the other two, and its time is short, so it is the best algorithm to use in this case. In terms of the algorithm's sustainability for the data, the metrics indicate that if the number of entries were to increase, AdaBoost would continue to provide a higher f-score and accuracy value. As the testing size grows, the metrics remain stable, indicating that the values would remain the same as the data grows.

0.7.2 Question 4 - Describing the Model in Layman's Terms

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

** HINT: **
When explaining your model, if using external resources please include all citations.
Answer:
Used this website: https://blog.paperspace.com/adaboost-optimizer/

AdaBoost is a classification algorithm that works sequentially to predict values. There are 3 weak learners that, when combined, build up off of each other to minimize error. The first weak learner is a decision tree, but it's a stump tree. It only has 1 parent node and its 2 children are leaves, meaning it only uses one variable to base its decision off of. The first weak learner classifies the training data, and each data sample has equal weights. There is a stump for each variable. The second learner then looks at how many data points are correctly and incorrectly labeled by each stump, and changes the weights of the samples depending on how they are classified, so the points incorrectly classfied have higher weight. Then the next time they are classified, they will be classified correctly. There is also a higher weight on classifiers that classify points more correctly, so higher weight means higher accuracy. The third learner builds on this and completes the same process until the maximum iteration is reached and the points are as accurately identified as possible.

0.7.3 Implementation: Model Tuning

Fine tune the chosen model. Use grid search (GridSearchCV) with at least one important parameter tuned with at least 3 different values. You will need to use the entire training set for this. In the code cell below, you will need to implement the following: - Import sklearn.grid_search.GridSearchCV and sklearn.metrics.make_scorer. - Initialize the classifier you've chosen and store it in clf. - Set a random_state if one is available to the same state you set before. - Create a dictionary of parameters you wish to tune for the chosen model. - Example: parameters = {'parameter' : [list of values]}. - **Note:** Avoid tuning the max_features parameter of your learner if that parameter is available! - Use make_scorer to create an fbeta_score scoring object (with $\beta = 0.5$). - Perform grid search on the classifier clf using the 'scorer', and store it in grid_obj. - Fit the grid search object to the training data (X_train, y_train), and store it in grid_fit.

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

```
In [12]: # TODO: Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
         from sklearn.model_selection import GridSearchCV
         from sklearn.metrics import make_scorer
         # TODO: Initialize the classifier
         clf = clf_B

         # 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" : [10,100,200,400], "learning_rate" : [0.001, 0.005, .01,

         # 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 usin
         grid_obj = GridSearchCV(clf, param_grid = parameters, scoring = scorer)

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

         # Get the estimator
         best_clf = grid_fit.best_estimator_

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

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

```
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)


Unoptimized model
------
Accuracy score on testing data: 0.8576
F-score on testing data: 0.7246


Optimized Model
```

```
------
Final accuracy score on the testing data: 0.8652
Final F-score on the testing data: 0.7401
```

0.7.4   Question 5 - Final Model Evaluation

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

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

Metric	Unoptimized Model	Optimized Model
Accuracy Score	0.8576	0.8652
F-score	0.7246	0.7401

Results: Answer:

   The optimized model's accuracy score is 0.8652. The optimized model's F-score is 0.7401. These scores are better than the unoptimized model, which makes sense since the optimized model selects the best features for the model to have. The optimized model's metrics are much higher from the naive predictor, as the naive predictor always predicts that an individual makes more than 50K

0.8   Feature Importance

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

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

0.8.1   Question 6 - Feature Relevance Observation

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

1. Education - amount of education is very a very important factor in determining the amount of money someone makes.
2. Occupation - some jobs tend to have higher salaries than others, leading to higher overall income
3. Hours-per-Week - the more hours someone works in a week, the more they would probably make
4. Native Country - 50K in dollars has a different conversion in different currencies depending on their own inflation levels, so an income of an individual might depend on the environment they live in.
5. Age - older people tend to have higher salaries after being in the workforce longer

0.8.2   Implementation - Extracting Feature Importance

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

In the code cell below, you will need to implement the following: - Import a supervised learning model from sklearn if it is different from the three used earlier. - Train the supervised model on the entire training set. - Extract the feature importances using `'.feature_importances_'`.

```
In [13]:  # TODO: Import a supervised learning model that has 'feature_importances_'


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


20
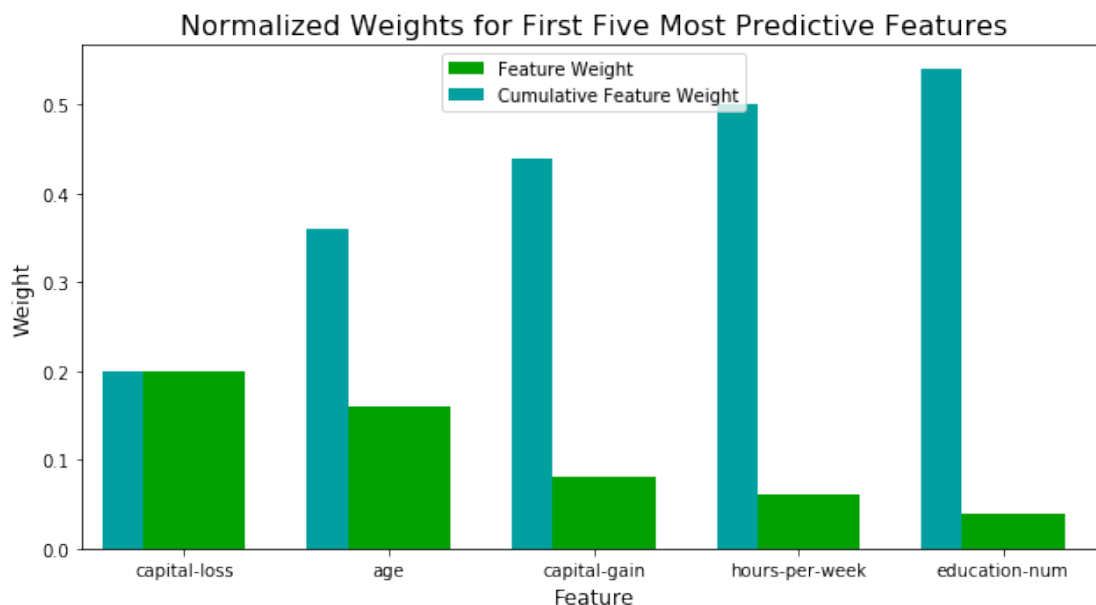
0.8.3 Question 7 - Extracting Feature Importance

Observe the visualization created above which displays the five most relevant features for predicting if an individual makes at most or above $50,000.
* How do these five features compare to the five features you discussed in **Question 6**? * If you were close to the same answer, how does this visualization confirm your thoughts? * If you were not close, why do you think these features are more relevant?

 Answer:

 I expected Age, Hours per Week, and Education to be key features that determine income. I did not expect, however, for capital gains and losses to have such a large impact, greater than that of Occupation and Native Country. I also did rank Age, Hours per Week incorrectly, but I did rank Education correctly, as it has the largest weight in determining income level. This visualization confirms the assumption that education and hours per week are important for determining income level, but I did not think age was as relevant as it is. Seeing the visualization, I guess it makes sense because the data show that the majority of older people do make above 50K and therefore it is a decisive feature. Additionally, with capital gains and losses, I wasn't sure how these features would indicate level of income. However, I can see that people with higher income tend to invest more, and therefore this will lead to a higher level of gains or losses from investments. Those with higher incomes will make greater capital gains and therefore that is an important factor in predicting income level, and with investing, the risk of losing capital is there too. Capital gains seems to have a greater importance than losses because gains can only be made when the investment is profitable, but losses can be made without having to make an investment, meaning that those below 50K who do not invest could also lose money.

0.8.4 Feature Selection

How does a model perform if we only use a subset of all the available features in the data? With less features required to train, the expectation is that training and prediction time is much lower — at the cost of performance metrics. From the visualization above, we see that the top five most important features contribute more than half of the importance of **all** features present in the data. This hints that we can attempt to *reduce the feature space* and simplify the information required for the model to learn. The code cell below will use the same optimized model you found earlier, and train it on the same training set *with only the top five important features*.

```
In [14]: # 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, be
        print("\nFinal Model trained on reduced data\n------")
        print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduced_predicti
        print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_predictions,
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8652
F-score on testing data: 0.7401


Final Model trained on reduced data
------
Accuracy on testing data: 0.8417
F-score on testing data: 0.7017
```

0.8.5   Question 8 - Effects of Feature Selection

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

Answer:
The Final Model trained on reduced data has less accuracy and a lower f-score as compared to the full data. This is becuase reducing the number of factors to only the 5 most important will lead to a greater number of incorrectly identified data points, but this has only gone down by a very small portion. If training time were a factor, I would consider using the reduced data as my training set, because training and testing such large amounts of data with 13 total factors takes up a lot of time and the accuracy and f-score values are not as much different as just when testing with 5.

Note: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to
File -> Download as -> HTML (.html). Include the finished document along with this notebook as your submission.

0.9   Before You Submit

You will also need run the following in order to convert the Jupyter notebook into HTML, so that your submission will include both files.

```
In [15]: !!jupyter nbconvert *.ipynb

Out[15]: ['[NbConvertApp] Converting notebook finding_donors.ipynb to html',
         '[NbConvertApp] Writing 507618 bytes to finding_donors.html']
```