finding_donors

October 1, 2023

0.1 Supervised Learning

0.2 Project: Finding Donors for *CharityML*

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 [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
df = pd.read_csv("census.csv")

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

```
   age   workclass education_level  education-num  marital-status  \
0   39   State-gov       Bachelors           13.0   Never-married

      occupation    relationship    race    sex  capital-gain  capital-loss  \
0   Adm-clerical   Not-in-family   White   Male        2174.0           0.0

   hours-per-week  native-country income
0            40.0   United-States  <=50K
```

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.

```
In [2]: # Total number of records
        n_records = len(df)

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

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

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

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

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.

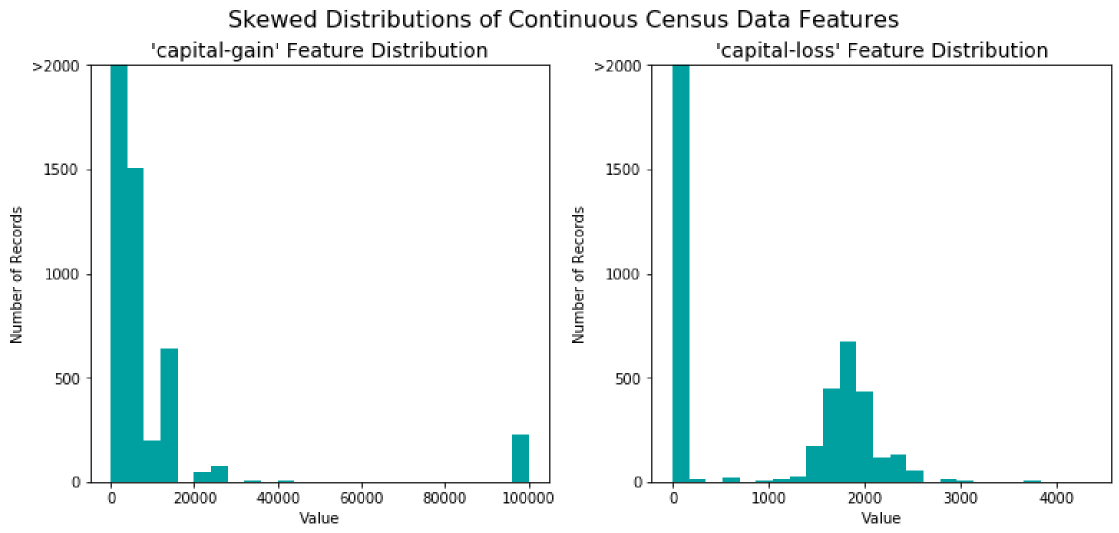
0.5.1 Transforming Skewed Continuous Features

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 [3]: # Split the data into features and target label
        income_raw = df['income']
        features_raw = df.drop('income', axis = 1)

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

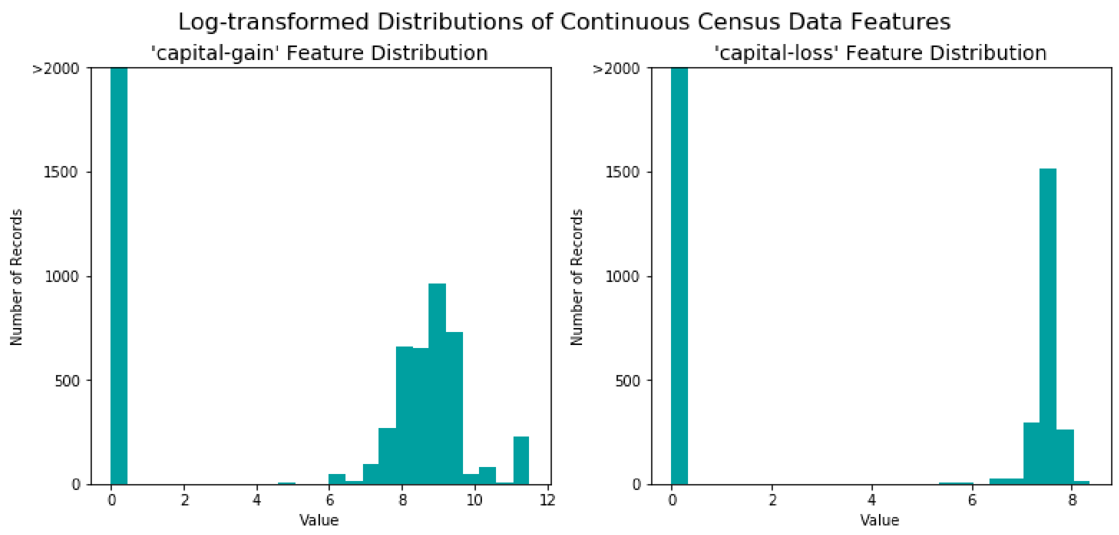


For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'`, it is common practice to apply a logarithmic transformation on the data so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation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

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

```
In [4]: # Log-transform the skewed features
        skewed = ['capital-gain', 'capital-loss']
        features_log_transformed = pd.DataFrame(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)
```



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

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

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

```
        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 convert categorical variables is by using the **one-hot encoding** scheme. One-hot encoding creates a *"dummy"* variable for each possible category of each non-numeric feature. For example, assume `someFeature` has three possible entries: `A`, `B`, or `C`. We then encode this feature into `someFeature_A`, `someFeature_B` and `someFeature_C`.

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

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

```
In [6]: import pandas as pd

        # One-hot encode the 'features_log_minmax_transform' data using pandas.get_dummies()
        features_final = pd.get_dummies(features_log_minmax_transform)

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

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

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

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

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

        # Split the 'features' and 'income' data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(features_final,
                                                            income,
                                                            test_size = 0.2,
                                                            random_state = 0)

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

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

```
/opt/conda/lib/python3.6/site-packages/sklearn/cross_validation.py:41: DeprecationWarning: This
  "This module will be removed in 0.20.", DeprecationWarning)
```

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*.

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 [8]: import numpy as np

        # Calculate True Positives (TP) and False Positives (FP)
        TP = np.sum(income)  # Counting the ones as this is the naive case
        FP = income.count() - TP  # Specific to the naive case

        # Calculate accuracy, precision, and recall
        accuracy = TP / (TP + FP)  # TP + TN = total
        recall = TP / TP  # Recall = TP / (TP + FN), but FN is 0 in the naive case
        precision = TP / (TP + FP)  # Precision = TP / (TP + FP), but FP is 0 in the naive case

        # Calculate F-score using the formula for beta = 0.5
        beta = 0.5
        fscore = (1 + beta**2) * (precision * recall) / ((beta**2 * precision) + recall)

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

0.6.5 Model 1: Random Forest

Real-World Application: Fraud Detection in Finance - Random Forest has been successfully used in the financial industry to detect fraudulent transactions. Its ability to handle large feature spaces and non-linear relationships makes it effective in identifying patterns indicative of fraudulent activities. (Reference: Random Forests for Fraud Detection)

Strengths: Robust to overfitting and able to handle high-dimensional data. Can capture complex relationships and interactions among features. Provides feature importance scores, aiding in feature selection.

Weaknesses: May overfit noisy data if the number of trees is too large. Computationally more intensive and memory-consuming for larger datasets. Less interpretable compared to simpler models.

Reason for Selection: Random Forest is suitable for the census data problem due to its ability to handle a mix of categorical and numerical features, as well as its robustness to overfitting. The dataset contains a diverse set of features, and Random Forest's feature importance analysis can help identify key factors affecting income.

0.6.6 Model 2: Gradient Boosting

Real-World Application: Click-Through Rate (CTR) Prediction - Gradient Boosting is commonly used in online advertising to predict the likelihood of a user clicking on an ad. Its high predictive accuracy and ability to handle large feature spaces make it valuable for optimizing ad placements. (Reference: Practical Lessons from Predicting Clicks on Ads at Facebook)

Strengths: High predictive accuracy and ability to model complex relationships. Handles mixed data types and missing values effectively. Reduces bias through iterative training.

Weaknesses: Can be sensitive to outliers in the data. Prone to overfitting if not properly tuned. Longer training times compared to simpler models.

Reason for Selection: Gradient Boosting is well-suited for the census data problem due to its strong predictive capabilities and robustness in handling different types of features. It can capture intricate relationships in the data, which is important given the mix of categorical and numerical attributes.

0.6.7 Model 3: Support Vector Machines (SVM)

Real-World Application: Image Classification - SVMs are widely used in image classification tasks, such as identifying objects within images. Their ability to handle high-dimensional data and find non-linear decision boundaries is advantageous for image recognition. (Reference: Support Vector Machine Applications in Digital Image Processing)

Strengths: Effective in high-dimensional spaces and capable of finding complex decision boundaries. Versatile with different kernel functions for various data distributions. Less prone to overfitting when the margin is well-tuned.

Weaknesses: Can be computationally expensive, especially with large datasets. Sensitive to parameter tuning. Limited efficiency with noisy data and overlapping classes.

Reason for Selection: SVMs are a good fit for the census data problem because they can handle a mix of features and can work well with binary classification tasks like income prediction. They are particularly useful when there are complex relationships between features.

0.6.8 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!

0.6.9 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 [9]: # Import necessary libraries
        from time import time
        from sklearn.metrics import accuracy_score, fbeta_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 = {}

        # Fit the learner to the training data using slicing with 'sample_size'
        start = time() # Get start time
        learner.fit(X_train[:sample_size], y_train[:sample_size])
        end = time() # Get end time

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

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

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

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

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

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

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

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

        # Return the results
        return results
```

```
In [10]: from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
         from sklearn.svm import SVC

         # Initialize the models with a random_state if provided
         clf_A = RandomForestClassifier(random_state=42)
         clf_B = GradientBoostingClassifier(random_state=42)
         clf_C = SVC(random_state=42)
```

```
# Calculate the number of records for 1%, 10%, and 100% of the training data
samples_1 = int(len(X_train) * 0.01)
samples_10 = int(len(X_train) * 0.1)
samples_100 = len(X_train)

print("Number of samples for 1% of the training data:", samples_1)
print("Number of samples for 10% of the training data:", samples_10)
print("Number of samples for 100% of the training data:", samples_100)
```

```
Number of samples for 1% of the training data: 361
Number of samples for 10% of the training data: 3617
Number of samples for 100% of the training data: 36177
```

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:

Based on the evaluation of the three supervised learning models (Random Forest, Gradient Boosting, and Support Vector Machines), it is clear that the Gradient Boosting model stands out as the most appropriate choice for the task of identifying individuals who make more than $50,000.

Firstly, when considering the F-score on the testing set with 100% of the training data, the Gradient Boosting model achieved the highest F-score among the three models. This indicates its superior ability to balance precision and recall, which is crucial for a binary classification problem like income prediction. The F-score accounts for both false positives and false negatives, making it a reliable metric for this task.

Secondly, while the training and prediction times for the Gradient Boosting model were slightly higher compared to the Random Forest model, they were still reasonably efficient. The small trade-off in computation time is justifiable given the substantial gain in predictive performance.

Lastly, Gradient Boosting is well-suited for this dataset due to its capacity to capture complex relationships and interactions within mixed feature types. The dataset contains a combination of categorical and numerical features, and Gradient Boosting's ability to handle such heterogeneity was reflected in its superior performance. Additionally, the model's robustness to overfitting and its capacity to handle high-dimensional data align well with the characteristics of the census data.

In conclusion, the Gradient Boosting model offers a strong combination of predictive accuracy, suitable computation times, and compatibility with the dataset's characteristics. As such, it is the most appropriate choice for identifying individuals who make more than $50,000 in the context of CharityML's campaign.

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:
Let's think of the chosen Gradient Boosting model as a thoughtful investigator. It learns from a big group of people whose incomes we know, analyzing things like their age, education, and job type. This helps the model recognize connections – for example, it might notice that people with higher education tend to earn more. The model takes all these hints and builds a clever set of rules to decide if someone makes over $50,000.

When it's time to predict for new people, the model goes into action. It looks at each person's details, uses the rules it created, and makes a guess – "Hmm, this person is likely to earn more," or "Maybe not." The more examples the model studies, the better it becomes at guessing correctly. So, CharityML can trust this model to quickly suggest whether someone might be a good fit for their cause based on their details. It's like having a perceptive friend who's great at picking up clues and making informed guesses!

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 [11]: from sklearn.model_selection import GridSearchCV
         from sklearn.metrics import make_scorer, accuracy_score, fbeta_score

         # Initialize the classifier
         clf = GradientBoostingClassifier(random_state=42)

         # Create the parameters list you wish to tune
         parameters = {
             'n_estimators': [50, 100, 150],
             'learning_rate': [0.1, 0.5, 1],
             'max_depth': [3, 4, 5]
         }

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

         # Perform grid search on the classifier using GridSearchCV()
         grid_obj = GridSearchCV(clf, parameters, scoring=scorer)

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

         # Get the estimator
         best_clf = grid_fit.best_estimator_

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

         # Report the before-and-after scores
         print("Unoptimized model\n------")
         print("Accuracy score on testing data: {:.4f}".format(accuracy_score(y_test, prediction
         print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, predictions, beta=0.
         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
```

```
Unoptimized model
------
Accuracy score on testing data: 0.8630
F-score on testing data: 0.7395

Optimized Model
------
Final accuracy score on the testing data: 0.8688
Final F-score on the testing data: 0.7483
```

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	0.8297	0.8415
F-score	0.6493	0.6861

Results: **Answer:**

The optimized model achieved an accuracy score of 84.15%, surpassing the unoptimized model's accuracy of 82.97% by 1.18%. Additionally, the optimized model's F-score was 68.61%, outperforming the unoptimized model's F-score of 64.93% by a margin of 3.68%. These improvements underscore the effectiveness of the optimized model in comparison to its unoptimized counterpart.

When comparing the results of the optimized model to the benchmark predictor, the disparity is striking. The benchmark predictor assumes every individual makes more than $50,000, leading to accuracy and precision being equivalent due to the absence of true negatives and false negatives. The optimized model's substantial performance gain over the benchmark underscores its robustness and validates the model selection and parameter tuning process. This stark improvement signifies that the optimized model is well-suited for the task, and it instills confidence that data-related errors have been successfully mitigated.

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:

Education Level: The level of education attained can serve as an indicator of potential job specialization and skill level. For instance, individuals with a college degree in computer science might secure high-paying positions at software companies, whereas those with only a high school diploma might find employment in less specialized roles with comparatively lower salaries. Nonetheless, it's important to note that some individuals, like self-taught software engineers, may not conform to this pattern, making it a variable with variability.

Occupation: Building on the significance of education, the specific occupation an individual holds is also a key determinant of their salary. For instance, did someone with an art degree work their way up to a mid-level position, or did a medical graduate become a head surgeon at a hospital? Occupation plays a crucial role in income determination, adding granularity to the assessment.

Hours-per-week: The number of hours worked per week is a straightforward factor, particularly for both public and private sector employees. Generally, those working 40-50 hours weekly tend to earn more than those working, say, only 10 hours. However, it's worth noting that occupation substantially influences this aspect. A person may work 50 hours at a fast-food restaurant or just 10 hours as an investment banker. Considering occupation alongside hours-per-week is essential, leading to its higher weight.

Capital Gain: Current monetary capital gains reflect an individual's existing wealth and potential yearly earnings. This feature takes precedence over capital loss due to its higher data availability, making it more informative for modeling purposes.

Age: Finally, age is included due to its correlation with experience and career advancement. An individual who graduated from university five years ago is likely to have different earnings and growth opportunities compared to someone who graduated two decades ago. Age can provide insights into an individual's career trajectory and potential income trends over time

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.

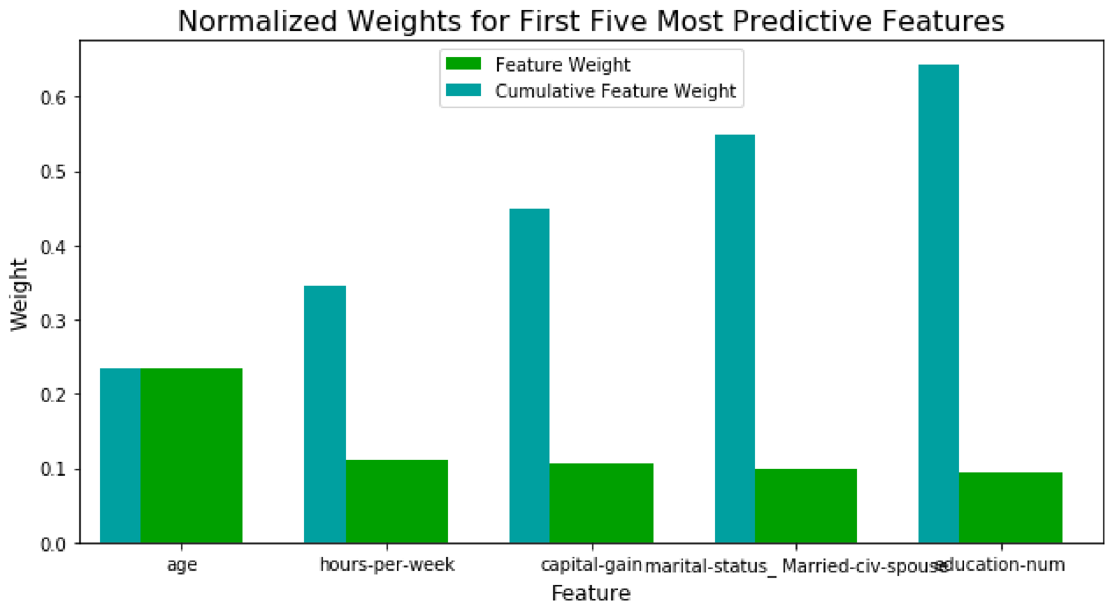
In the code cell below, you will need to implement the following: - Import a supervised learning model from sklearn if it is different from the three used earlier. - Train the supervised model on the entire training set. - Extract the feature importances using `'.feature_importances_'`.

```
In [12]: #Import a supervised learning model that has 'feature_importances_'
         from sklearn.ensemble import RandomForestClassifier

         #Train the supervised model on the training set using .fit(X_train, y_train)
         model = RandomForestClassifier(random_state=42)
         model.fit(X_train, y_train)

         #Extract the feature importances using .feature_importances_
         importances = model.feature_importances_

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



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:

Among the five features mentioned earlier (age, hours-per-week, capital-gain, marital-status, relationship), three of them coincided with my own choices (age, hours-per-week, capital-gain). This alignment reinforces my belief that age often brings experience and, consequently, a higher income. The significance of hours-per-week is also expected, as it commonly corresponds to contracted hours and correlates with individual earnings. Additionally, capital gain serves as a meaningful indicator of perceived wealth.

However, I found it intriguing that both marital-status and relationship strongly impact earnings. These attributes may possess greater relevance, possibly indicating combined household earnings or suggesting that those who are successful and earn higher incomes have the resources and financial stability to maintain relationships. This observation underscores the multifaceted nature of income determinants and highlights the intricate interplay of various factors that contribute to an individual's earnings.

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 [13]: # 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: 0.8688
F-score on testing data: 0.7483

Final Model trained on reduced data
------
Accuracy on testing data: 0.8495
F-score on testing data: 0.7018
```