Data Scientist Nanodegree

Supervised Learning

Project: Finding Donors for *CharityML*

Welcome to the first project of the Data Scientist Nanodegree! In this notebook, some template code has already been provided for you, and it will be your job to implement the additional functionality necessary to successfully complete this project. Sections that begin with **'Implementation'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a `'TODO'` statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Please specify WHICH VERSION OF PYTHON you are using when submitting this notebook. Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Getting Started

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

The dataset for this project originates from the [UCI Machine Learning Repository](). The 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.

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.

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318

Enter your authorization code:
··········
Mounted at /content/gdrive
```

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import mean_squared_error
import time
#importing machine learning libraries
import pickle
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import classification_report

# for hyperparamter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold, GridSearchCV, StratifiedKFold
from sklearn.metrics import accuracy_score, make_scorer
from sklearn.feature_selection import SelectKBest, chi2

#importing classifiers
from lightgbm import LGBMClassifier
```

```
from lightgbm import plot_importance
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
# pandas profilling for fast data understanding
from pandas_profiling import ProfileReport

plt.style.use( 'ggplot') # the plots style

# fixing random seed for consistency
random_state = 1

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

```
# Load the Census dataset
data = pd.read_csv('/content/gdrive/My Drive/Udacity Nano-degree projects data/L1.census.csv'
submission = pd.read_csv('/content/gdrive/My Drive/Udacity Nano-degree projects data/L1.examp
# Success - Display the first 5 record
data.head()
```

	age	workclass	education_level	education-num	marital-status	occupation	relationship	ra
0	39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	Wh
1	50	Self-emp-not-inc	Bachelors	13.0	Married-civ-spouse	Exec-managerial	Husband	Wh
2	38	Private	HS-grad	9.0	Divorced	Handlers-cleaners	Not-in-family	Wh
3	53	Private	11th	7.0	Married-civ-spouse	Handlers-cleaners	Husband	Bla
4	28	Private	Bachelors	13.0	Married-civ-spouse	Prof-specialty	Wife	Bla

```
submission.head()
```

id	income

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.

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

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

```
# TODO: Total number of records
n_records = data.count()[0]

# TODO: Number of records where individual's income is more than $50,000
n_greater_50k = data.income.value_counts()[1]

# TODO: Number of records where individual's income is at most $50,000
n_at_most_50k = data.income.value_counts()[0]

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

# Print the results
print("Total number of records: {}".format(n_records))
print("Individuals making more than $50,000: {}".format(n_greater_50k))
print("Individuals making at most $50,000: {}".format(n_at_most_50k))
print("Percentage of individuals making more than $50,000: {}%".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: 0.2478439697492371%
```

** Featureset Exploration **

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

Preparing the Data

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

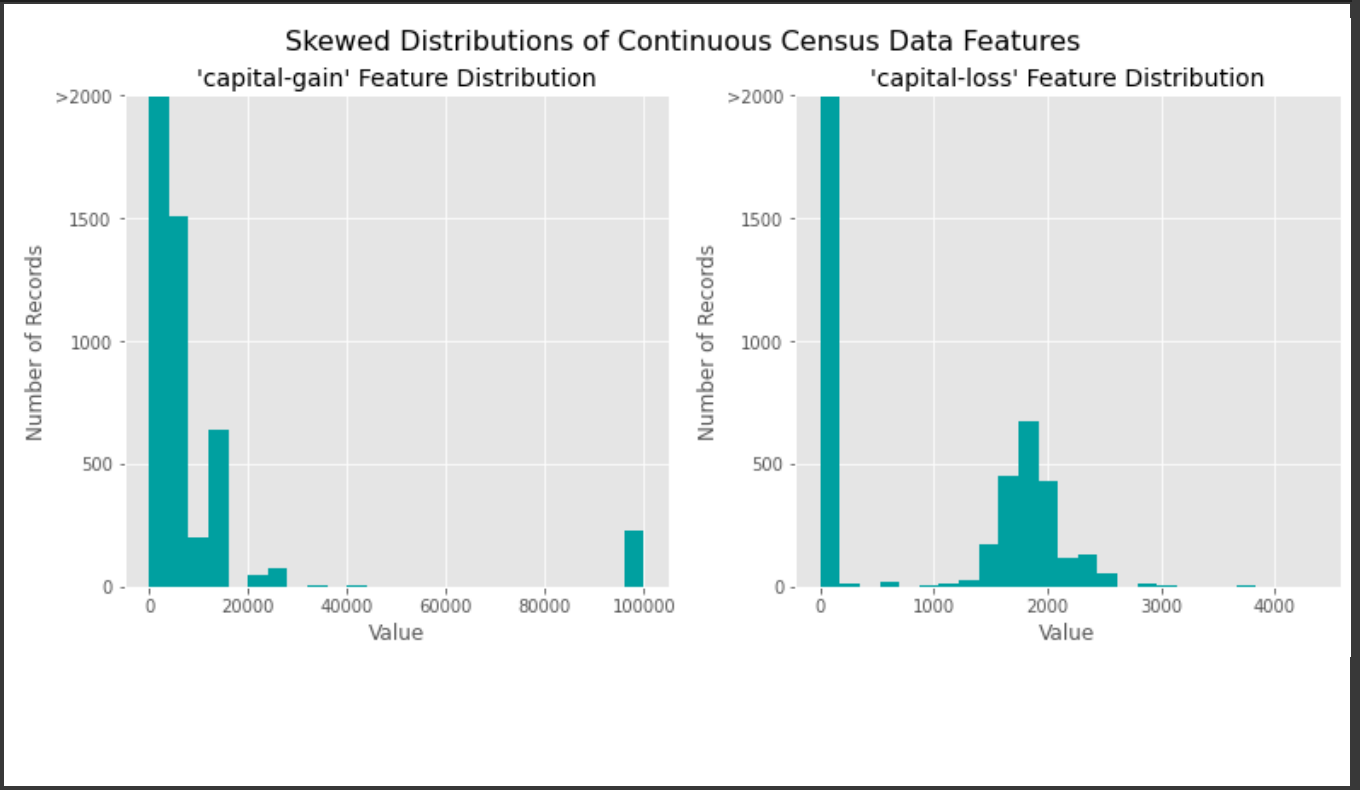
Transforming Skewed Continuous Features

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

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

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

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



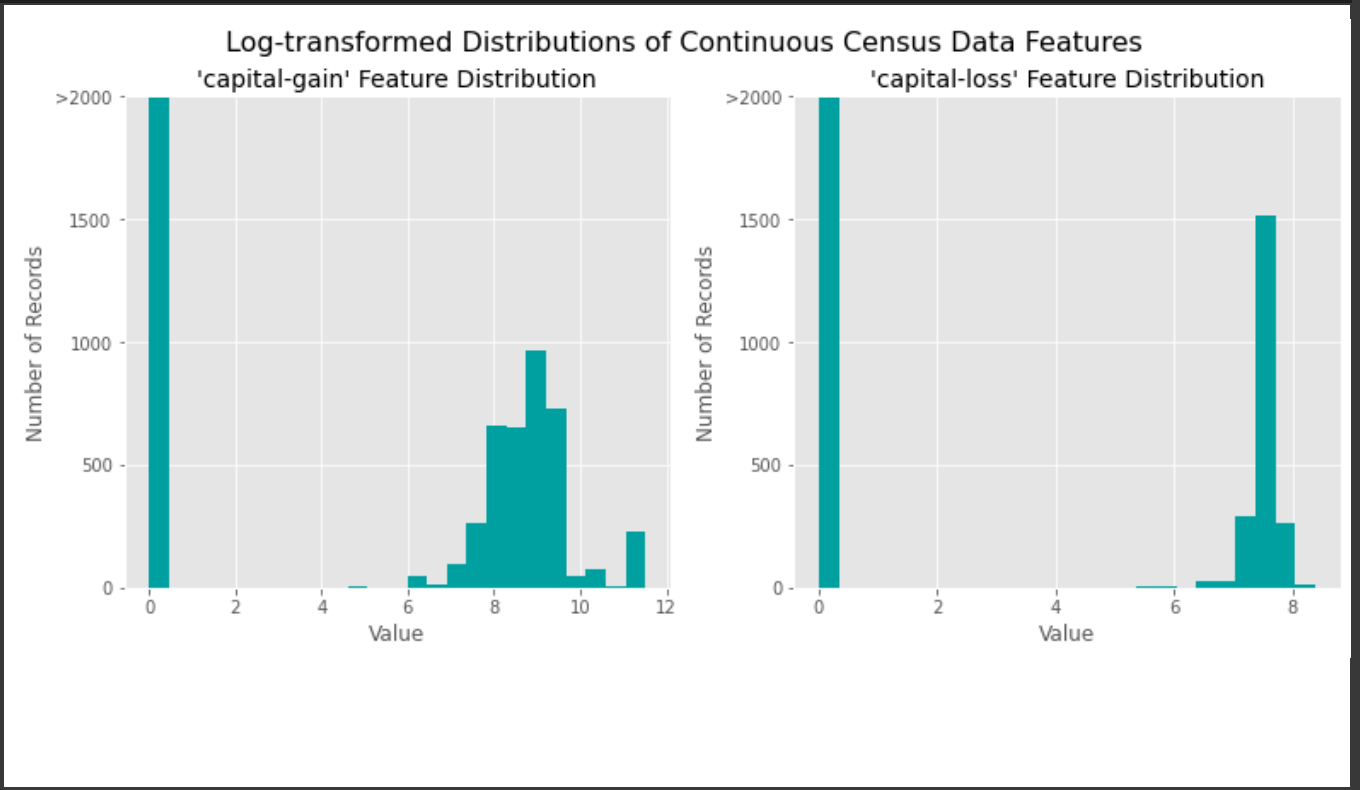
For highly-skewed feature distributions such as `'capital-gain'` and `'capital-loss'`, it is common practice to apply a [logarithmic transformation] on the data so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation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

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

```
# Log-transform the skewed features
skewed = ['capital-gain', 'capital-loss']
features_log_transformed = features_raw.copy() # i changed this line like this because the ou
features_log_transformed[skewed] = features_raw[skewed].apply(lambda x: np.log(x + 1))

# Visualize the new log distributions
```

```
# Visualize the new log distributions
vs.distribution(features_log_transformed, transformed = True)
```



Normalizing Numerical Features

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

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

```
# 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[nume

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

	age	workclass	education_level	education-num	marital-status	occupation	relationship
0	0.301370	State-gov	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family
1	0.452055	Self-emp-not-inc	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband
2	0.287671	Private	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family
3	0.493151	Private	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband

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`.

```
# TODO: One-hot encode the 'features_log_minmax_transform' data using pandas.get_dummies()

cat_cols = list(features_log_minmax_transform.select_dtypes(include = ['object']).columns) #
one_hot_encoded_features = pd.get_dummies(features_log_minmax_transform[cat_cols]) # Encodin
```

```
numerical_features = features_log_minmax_transform[numerical] # Get me numerical columns only
features_final = pd.concat([one_hot_encoded_features , numerical_features], axis = 1)
print(features_final.head())

# TODO: Encode the 'income_raw' data to numerical values
income = income_raw.apply( lambda x : 1 if x == '>50K'  else 0)
print('\n\n' , 'binary target variable : ', income , '\n\n')

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

print(encoded)
```

```
   workclass_ Federal-gov  workclass_ Local-gov  ...  capital-loss  hours-per-week
0                       0                     0  ...           0.0        0.397959
1                       0                     0  ...           0.0        0.122449
2                       0                     0  ...           0.0        0.397959
3                       0                     0  ...           0.0        0.397959
4                       0                     0  ...           0.0        0.397959

[5 rows x 103 columns]


 binary target variable :  0        0
1        0
2        0
3        0
4        0
        ..
45217    0
45218    0
45219    0
45220    0
45221    1
Name: income, Length: 45222, dtype: int64


103 total features after one-hot encoding.
['workclass_ Federal-gov', 'workclass_ Local-gov', 'workclass_ Private', 'workclass_ Se
```

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.

```
# 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

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

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

Evaluating Model Performance

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

Metrics and the Naive Predictor

CharityML, equipped with their research, knows individuals that make more than \$50,000 are most likely to donate to their charity. Because of this, *CharityML* is particularly interested in predicting who makes more than \$50,000 accurately. It would seem that using **accuracy** as a metric for evaluating a particular model's 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 **$F_{0.5}$ 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).

X_train.count()[0]X_train.count()[0]X_XxxxfeeeeX_train.count()[0]X_train.count()[0]ssftime.time()dfvd### 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.

```
TP = np.sum(income) # Counting the ones as this is the naive case.
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 = (TP + TN) / (TP + FP + TN + FN)
recall = TP / (TP + FN)
precision = TP / (TP + FP)

# TODO: Calculate F-score using the formula above for beta = 0.5 and correct values for preci
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, fscore))
```

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

Supervised Learning Models

The following are some of the supervised learning models that are currently available in `scikit-learn` that you may choose from:

- Gaussian Naive Bayes (GaussianNB)
- Decision Trees
- Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting)
- K-Nearest Neighbors (KNeighbors)
- Stochastic Gradient Descent Classifier (SGDC)
- Support Vector Machines (SVM)
- Logistic Regression

Question 2 - Model Application

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

- Describe one real-world application in industry where the model can be applied.

- What are the strengths of the model; when does it perform well?
- What are the weaknesses of the model; when does it perform poorly?
- What makes this model a good candidate for the problem, given what you know about the data?

** HINT: **

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

Single Decision Tree:

- Predicting high occupancy dates for hotels
- 1) Can give us insights and points to how important are the features 2) Efficient and simple in its implementation and excustion time 3) Does not suffer from outliers and as linear models so we don't need to normalize and scale the numerical features 4) dos not effected by corrleated features like numerical models like linear regression for example 5) can capture non linear patterns ... it's best to use when you want scalable and fast model on the cost some accuracy, also when you don't want your model to overfit.
- 1) They are unstable, so if a small change happens in the data a big impact will happen to the tree 2) you know when using decision tree that you will lose some accuracy compared with other sophisticated models 3) overfit and perform poorly when the categorical variable is one-hot encoded and has many levels ... should not be used when : 1) accuracy is a priority 2) data is unstable and unreliable 3) most of features are categorical with many levels
- I will use this model as an initial-simple model to see its performance in comparison with a linear model and more complex ensamble technique to see which model is more fitted to the data

Resources

Logistic Regression :

- I personally used it to predict if a person has a Cardiovascular Disease or not Here
- 1) Very efficient, simple and fast 2) Does not require too many computational resources 3) it's highly interpretable 4) it doesn't require input features to be scaled 5) it doesn't require any tuning ... it's a really a good model to start with for the above reasons also to see if the data will be fitted linearly or need non linear models
- 1) Can not capture non-linear pattern 2) is effected badly when entring non impoertnant features or corrleated features unlike trees 3) can overfit easily ... do not use when you your

data can not be seprated with a line, a lot of noise in your data as it;s easliy overfitted 3) accuracy is not your priority

- To test seprating the data linearly

LightGBM :

- I personally used it in my graduation project to predict fture sales for a gaint tech company, Check it Here
- 1) Very fast compared to other ensamble technique (about 10 times faster than XGBoost) yet give very good results 2) dose not effected by redundant or unimportant features 3) high accuracy 4) give insights and features importance .... use it when you want to use ensamble model on a large amount of data it will be very fast, also when inisghts and accuracy are priorities , when the data has non-linear relationships
- 1) can easily overfit 2) time intensive compared to simple decision tree or simple linear models ... do not use when simplicity and scalability are priorities also when the data can be captured linearly then no need fo ensamble models
- To see how a complex boosting technique will fit on the data as if it has non-linear patterens will be captured

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!

```
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 = {}

    # TODO: Fit the learner to the training data using slicing with 'sample_size' using .fit(
    start = time.time() # Get start time
    learner = Learner
    learner.fit(X_train[:sample_size] , y_train[ : sample_size]  )
    end = time.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 .predict()
    start = time.time() # Get start time
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train[ : 300])
    end = time.time() # Get end time

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

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

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

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

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

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

    # Return the results
    return results
```

Implementation: Initial Model Evaluation

In the code cell, you will need to implement the following:

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

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

```
# TODO: Initialize the three models
clf_A = DecisionTreeClassifier(random_state = random_state )
clf_B = LogisticRegression(random_state = random_state , n_jobs = -1)
clf_C = LGBMClassifier(random_state = random_state , n_jobs = -1)

# TODO: Calculate the number of samples for 1%, 10%, and 100% of the training data

samples_100 = X_train.count()[0]
samples_10 = int(samples_100 * 0.1)
samples_1 = int(samples_100 * 0.01)

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


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

```
DecisionTreeClassifier trained on 361 samples.
DecisionTreeClassifier trained on 3617 samples.
DecisionTreeClassifier trained on 36177 samples.
LogisticRegression trained on 361 samples.
LogisticRegression trained on 3617 samples.
LogisticRegression trained on 36177 samples.
LGBMClassifier trained on 361 samples.
LGBMClassifier trained on 3617 samples.
LGBMClassifier trained on 36177 samples.
/content/visuals.py:121: UserWarning: Tight layout not applied. tight_layout cannot mak
  pl.tight_layout()
```


```
results['LGBMClassifier'][2]
```

```
{'acc_test': 0.8715312327252626,
 'acc_train': 0.88,
 'f_test': 0.7598014385573904,
 'f_train': 0.7352941176470589,
 'pred_time': 0.06342482566833496,
 'train_time': 0.6343479156494141}
```

we need to enhance this score `f_test: 0.7598014385573904`

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

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.

LightGBM

- We saw that the training time of LightGBM is very effecient in training time compared to logistic regression.
- Although Decision tree was better in the training accuracy and fscore but it was the worst in predicting which may be a sign of overfitting unlike LightGBM which showed moderate learning and predicting accuracy and fscore.
- predicting time was not good compared to the other 2 models but we can tolerate with that as all of them took secondes .
- The Algorithm is suitible from many prospective like stability on redundant and correlated columns, capturing non-linear patterns and effecincy on training time that would allow me to try different features and conduct many experiments without waiting for ever.

Question 4 - Describing the Model in Layman's Terms

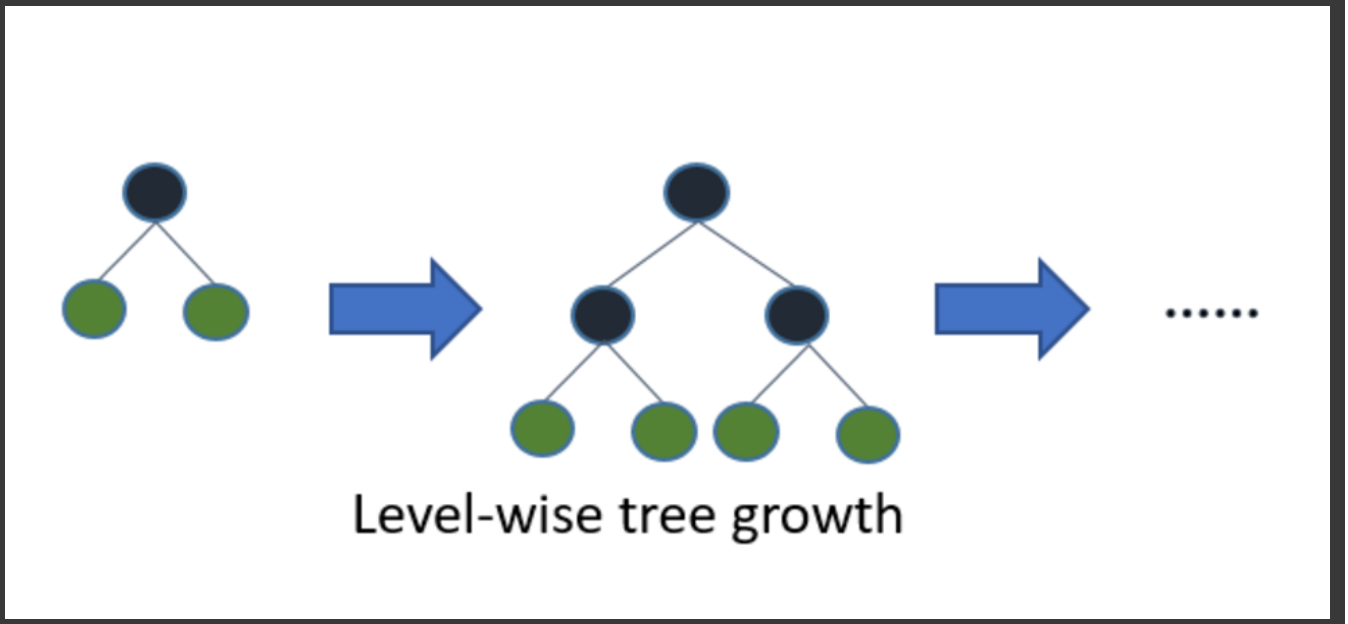
- In one to two paragraphs, explain to *CharityML*, in layman's terms, how the final model chosen is supposed to work. Be sure that you are describing the major qualities of the model, such as how the model is trained and how the model makes a prediction. Avoid using advanced mathematical jargon, such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

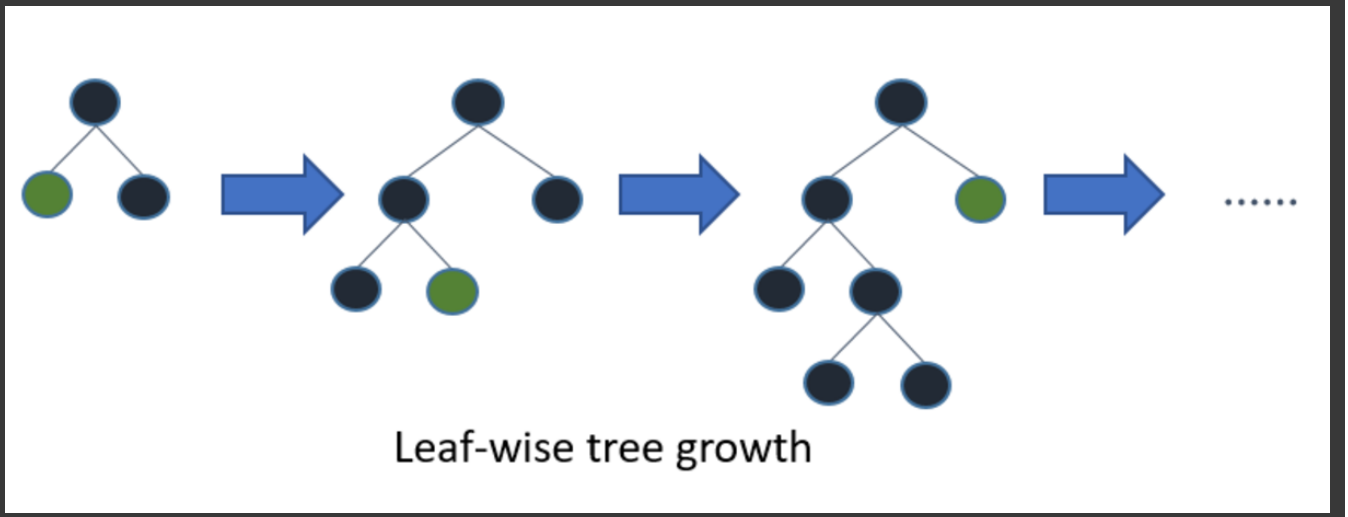
** HINT: **

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

- Light GBM is a gradient boosting framework that uses tree based learning algorithm.

- Light GBM grows tree vertically while other algorithm grows trees horizontally which generally makes it very efficient and in some cases get more accurate results
- In the following we will see regular boosting technique vs lightgbm which is leaf wise





- The general idea of boosting techniques generally is to use many weak learners connecting them togeather so that each new learner will benefit form the errors and predictions of the past learners, then by the end of the learning we will combine them all to make 1 strong model that kept adjusting the errors of the past weak learning through knwoing the oprimal set of wieghts.
- predicting then will be conducted after making the weights of the weak learners and the equation that capture the effect of each feature on the target variable (features coefficient)

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!

Note : I saved the model after the hyper paramaters tuning so you don't need to run the grid search just scroll down to the saved model using pickle.

```
clf = LGBMClassifier(random_state = random_state , n_jobs = -1 )

# TODO: Create the parameters list you wish to tune, using a dictionary if needed.
# HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, value2]}
# Define our search space for grid search
parameters = [
  {
              'n_estimators': [ 200 , 300 , 400, 500 ],
              'min_data_in_leaf': [2,35,100, 200],
              'bagging_fraction': [0.8,0.9,1],
              'learning_rate': [ 0.1 , 0.2 ],
              'num_leaves': [2,35,100 , 200],
              'verbose':[0]
  }
]
# Define cross validation
kfold = StratifiedKFold(n_splits=4)
# Scoring metric
scorer = make_scorer( fbeta_score, beta = 0.5 )

# Define grid search
grid_obj = GridSearchCV(
  clf,
```

```
  param_grid = parameters,
  cv=kfold,
  scoring = scorer,
  verbose = 1,
  n_jobs = -1,
  refit = True
)

# Get the estimator
best_clf = grid_obj.fit(X_train , y_train) # because i'm already using refit on the best mode

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

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

```
Fitting 4 folds for each of 384 candidates, totalling 1536 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed:  1.0min
[Parallel(n_jobs=-1)]: Done 196 tasks      | elapsed:  5.9min
[Parallel(n_jobs=-1)]: Done 446 tasks      | elapsed: 13.1min
[Parallel(n_jobs=-1)]: Done 796 tasks      | elapsed: 23.3min
[Parallel(n_jobs=-1)]: Done 1246 tasks      | elapsed: 36.7min
[Parallel(n_jobs=-1)]: Done 1536 out of 1536 | elapsed: 45.8min finished
Unoptimized model
------
Accuracy score on testing data: 0.8715
F-score on testing data: 0.7598

Optimized Model
------
Final accuracy score on the testing data: 0.8712
Final F-score on the testing data: 0.7590
```

Sometimes when we divide the learning rate by a constant and multiply the number of estimators by the same constant the accuracy get better , so i will try this trick where the constant = `4`

```
best_clf.best_params_
```

```
{'bagging_fraction': 0.8,
 'learning_rate': 0.1,
```

```
lgbm = LGBMClassifier(random_state = random_state , n_jobs = -1 , bagging_fraction= 0.8 , lea
                      n_estimators= (200*4) , num_leaves= 35, )
lgbm.fit(X_train , y_train)
best_predictions2 = lgbm.predict(X_test)
print("Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test, best_p
print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predictions
```

```
Final accuracy score on the testing data: 0.8721
Final F-score on the testing data: 0.7611
```

```
# saving the model to save the time of grid search
filename = 'LGBM_after_hyperparamater_tuning.sav'
pickle.dump(lgbm,open(filename,'wb'))
```

```
# to load the model from the disk :
lgbm = pickle.load(open('/content/LGBM_after_hyperparamater_tuning.sav','rb'))
best_predictions2 = lgbm.predict(X_test)
print("Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test, best_p
print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predictions
```

```
Final accuracy score on the testing data: 0.8721
Final F-score on the testing data: 0.7611
```

Question 5 - Final Model Evaluation

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

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

Results:

Metric	Unoptimized Model	Optimized Model
Accuracy Score	0.8715	0.8721
F-score	0.7598	0.7611

1. `Final accuracy, F-score score on the testing data: 0.8721 , 0.7611`
2. it's better
3. it's much better better --> from `0.2917` to `0.7611`

Feature Importance

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

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

Question 6 - Feature Relevance Observation

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

1. **capital-gain** --> the more capital gain the more likely income will be > 50k
2. **capital-loss** --> vice versa with **capital-gain**
3. **occupation** --> managers get paied more than employees for example or a pilot will take more than a regular worker and so on
4. **education_level** --> mabye the higher your educational level the more tolerated and awared with people need help
5. **hours-per-week** --> the more you work the higher you will be paid

Implementation - Extracting Feature Importance

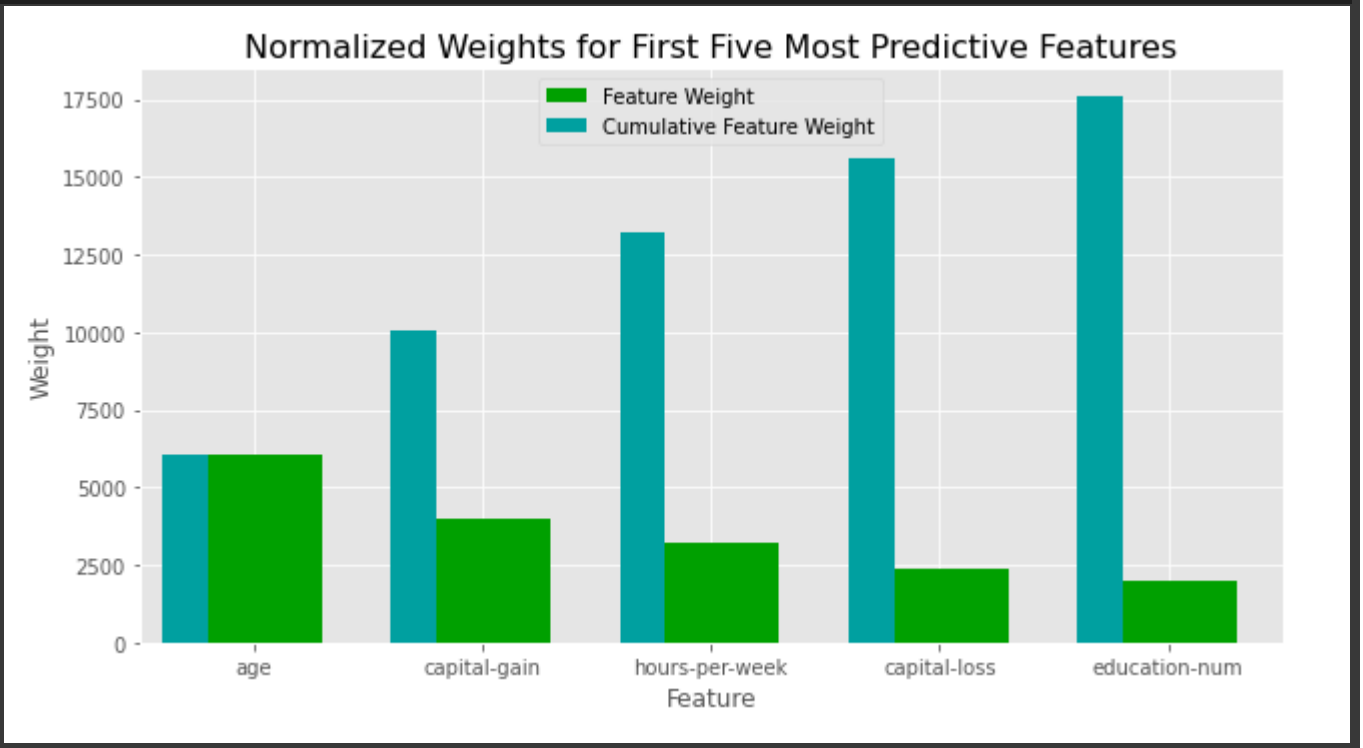
Choose a `scikit-learn` supervised learning algorithm that has a `feature_importance_` attribute availble for it. This attribute is a function that ranks the importance of each feature when making predictions based on the chosen algorithm.

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_'`.

```
importances = lgbm.feature_importances_

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



Question 7 - Extracting Feature Importance

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

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

1. 3 of them were right , instead of the education-num i chosed the education level and i missed the age for the sake of working hours
2. it's clear from the graph that these are the most important features based on their weights and cumulative weight
3. i didn't thought that age may have something to do in charity affairs but mabye older people get paied more so the probability increase when you get old

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*.

```
# 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(lgbm)).fit(X_train_reduced, y_train)

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

# Report scores from the final model using both versions of data
print("Final Model trained on full data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, best_predictions)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, beta = 0
print("\nFinal Model trained on reduced data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduced_predictions))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_predictions, beta
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8712
F-score on testing data: 0.7590

Final Model trained on reduced data
------
Accuracy on testing data: 0.8404
F-score on testing data: 0.7034
```

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?

The accuracy decreased a lot and since the model runs in seconds on the whole data set we don't have to sucrafice the accuracy, so i will recommend using the model on the whole data set.