Website (https://www.coleridgeinitiative.org)

Rayid Ghani, Frauke Kreuter, Julia Lane, Adrianne Bradford, Alex Engler, Nicolas Guetta Jeanrenaud, Graham Henke, Daniela Hochfellner, Clayton Hunter, Brian Kim, Avishek Kumar, Jonathan Morgan, and Benjamin Feder.

Citation to be updated on notebook export

Machine Learning

Introduction

In this tutorial, we will utilize our training and testing tables we created in the Machine Learning Preparation (ML_Data_Prep.ipynb) notebook to discuss how to build, evaluate, compare, and select models and how to reasonably and accurately interpret model results. You'll also get hands-on experience using the `scikit-learn` package in Python to model the `ml_model_train` and `ml_model_test` data tables we created in the `iri_usda_2019_db` Athena database.

This tutorial is based on chapter 6 of Big Data and Social Science (https://github.com/BigDataSocialScience/).

Glossary of Terms

There are a number of terms specific to machine learning that you will find repeatedly in this notebook.

- **Learning**: In machine learning, you'll hear about "learning a model." This is what you probably know as *fitting* or *estimating* a function, or *training* or *building* a model. These terms are all synonyms and are used interchangeably in the machine learning literature.
- **Examples**: These are what you probably know as *data points*, *observations*, or *rows*.
- **Features**: These are what you probably know as *independent variables*, *attributes*, *predictors*, or *explanatory variables.*
- **Underfitting**: This happens when a model is too simple and does not capture the structure of the data well enough.
- **Overfitting**: This happens when a model is too complex or too sensitive to the noise in the data; this can result in poor generalization performances or little applicability of the model to new data.
- **Regularization**: This is a general method to avoid overfitting by applying additional constraints to the model. For example, you can limit the number of features present in the final model or the weight coefficients applied to the (standardized) features.
- **Supervised learning**: This involves problems with one target or outcome variable (continuous or discrete) that we want to predict or classify data into. Classification, prediction, and regression fall into this category. We call the set of explanatory variables X **features**, and the outcome variable of interest Y the **label**.
- **Unsupervised learning**: This involves problems that do not have a specific outcome variable of interest, but rather we are looking to understand "natural" patterns or groupings in the data - looking to uncover some structure that we do not know about a priori. Clustering is the most common example of unsupervised learning, and another example is principal components analysis (PCA).

Python Setup

Before we begin, run the code cell below to initialize the libraries we'll be using in this assignment. We're already familiar with `pandas` , and `pyathenajdbc` from previous tutorials. Here we'll also be using `scikit-learn` (http://scikit-learn.org) to fit modeling.

In [ ]:
```
# Basic data analysis and visualization tools
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# SQL Connection
from pyathenajdbc import connect

# Machine learning tools
from sklearn.metrics import confusion_matrix, accuracy_score, precision_
score, recall_score, precision_recall_curve,roc_curve, auc
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifi
er,
                              GradientBoostingClassifier,
                              AdaBoostClassifier)
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier

# Seaborn settings for nicer graphs
sns.set_style("white")
sns.set_context("poster", font_scale=1.25, rc={"lines.linewidth":1.25,
"lines.markersize":8})
```

In [ ]:
```
# Database connection
conn = connect(s3_staging_dir = 's3://usda-iri-2019-queryresults/',
               region_name = 'us-gov-west-1',
               LogLevel = '0',
               workgroup = 'workgroup-iri_usda')
```

The Machine Learning Process

Let's recall the machine learning process:

- **Understand the problem and goal.** *This sounds obvious but is often nontrivial.* Problems typically start as vague descriptions of a goal - improving health outcomes, increasing graduation rates, understanding the effect of a variable *X* on an outcome *Y*, etc. It is really important to work with people who understand the domain being studied to dig deeper and define the problem more concretely. What is the analytical formulation of the metric that you are trying to optimize?
- **Formulate it as a machine learning problem.** Is it a classification problem or a regression problem? Is the goal to build a model that generates a ranked list prioritized by risk, or is it to detect anomalies as new data come in? Knowing what kinds of tasks machine learning can solve will allow you to map the problem you are working on to one or more machine learning settings and give you access to a suite of methods.
- **Data exploration and preparation.** Next, you need to carefully explore the data you have. What additional data do you need or have access to? What variable will you use to match records for integrating different data sources? Which variables exist in the data set? Are they continuous or categorical? What about missing values? Can you use the variables in their original form, or do you need to alter them in some way?
- **Feature engineering.** In machine learning language, what you might know as independent variables or predictors or factors or covariates are called "features." Creating good features is probably the most important step in the machine learning process. This involves doing transformations, creating interaction terms, or aggregating over data points or over time and space.
- **Method selection.** Having formulated the problem and created your features, you now have a suite of methods to choose from. It would be great if there were a single method that always worked best for a specific type of problem. Typically, in machine learning, you take a variety of methods and try them, empirically validating which one is the best approach to your problem.
- **Evaluation.** As you build a large number of possible models, you need a way choose the best among them. We'll cover methodology to validate models on historical data and discuss a variety of evaluation metrics. The next step is to validate using a field trial or experiment.
- **Deployment.** Once you have selected the best model and validated it using historical data as well as a field trial, you are ready to put the model into practice. You still have to keep in mind that new data will be coming in and that the model might change over time.

You're probably used to fitting models in physical or social science classes. In those cases, you probably had a hypothesis or theory about the underlying process that gave rise to your data, chose an appropriate model based on prior knowledge and fit it using least squares, and used the resulting parameter or coefficient estimates (or confidence intervals) for inference. This type of modeling is very useful for *interpretation*.

In machine learning, our primary concern is *generalization*. This means that:

- **We care less about the structure of the model and more about the performance** This means that we'll try out a whole bunch of models at a time and choose the one that works best, rather than determining which model to use ahead of time. We can then choose to select a *suboptimal* model if we care about a specific model type.
- **We don't (necessarily) want the model that best fits the data we've *already seen*,** but rather the model that will perform the best on *new data*. This means that we won't gauge our model's performance using the same data that we used to fit the model (e.g., sum of squared errors or R^2), and that "best fit" or accuracy will most often *not* determine the best model.

- **We can include a lot of variables in to the model.** This may sound like the complete opposite of what you've heard in the past, and it can be hard to swallow. But we will use different methods to deal with many of those concerns in the model fitting process by using a more automatic variable selection process.

Guiding Question

Recall our guiding question for this notebook:

Out of low-income households, can we predict which ones did not purchase a 100% whole wheat product in a year's time? If so, what are the most important household features?

> Note: For the purposes of this notebook, we just focused on 100% whole wheat purchases since they are the most common whole grain products (ers.usda.gov/webdocs/publications/93651/err-268.pdf?v=1165.6) consumed. If you would like to expand your analysis to include other whole grain options, such as oatmeal and corn tortillas, you need to add additional `where` clauses to include the products in the code where we created `ml_aggregate`. We can help you with the implementation if you are having trouble.

Data Exploration and Preparation in Python

During the first classes, we explored the data, linked different data sources, and created new variables. We will now implement our machine learning model on `ml_model_train` and test on `ml_model_test`.

1. **Creating labels**: Labels are the dependent variables, or *Y* variables, that we are trying to predict. In the machine learning framework, labels are often *binary*: true or false, encoded as 1 or 0. This outcome variable is aptly named `label`, and tracks the presence of a 100% whole wheat purchase for a low-income household in 2015 or 2016 for our training and testing sets, respectively.
2. **Decide on features**: Our features are our independent variables or predictors. Good features make machine learning systems effective. The better the features the easier it is the capture the structure of the data. You generate features using domain knowledge. In general, it is better to have more complex features and a simpler model rather than vice versa. Keeping the model simple makes it faster to train and easier to understand rather then extensively searching for the "right" model and "right" set of parameters. Machine learning algorithms learn a solution to a problem from sample data. The set of features is the best representation of the sample data to learn a solution to a problem.

> Refer to the ML_Data_Prep.ipynb (ML_Data_Prep.ipynb) notebook for how the features and labels were created.

1. **Feature engineering** is the process of transforming raw data into features that better represent the underlying problem/data/structure to the predictive models, resulting in improved model accuracy on unseen data." (from Discover Feature Engineering (http://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/)). In text, for example, this might involve deriving traits of the text like word counts, verb counts, or topics to feed into a model rather than simply giving it the raw text. Other examples of feature engineering include:

    - **Transformations**, such a log, square, and square root.
    - **Dummy (binary) variables**, sometimes known as *indicator variables*, often done by taking categorical variables (such as industry) that do not have a numeric value and adding them to models as a binary value.
    - **Discretization**. Several methods require features to be discrete instead of continuous. This is often done by binning, which you can do by equal width, deciles, Fisher-Jenks, etc.
    - **Aggregation.** Aggregate features often constitute the majority of features for a given problem. These use different aggregation functions (*count, min, max, average, standard deviation, etc.*) which summarize several values into one feature, aggregating over varying windows of time and space. For example, we may want to calculate the *number* (and *min, max, mean, variance*, etc.) of crimes within an *m*-mile radius of an address in the past *t* months for varying values of *m* and *t*, and then use all of them as features.
2. **Cleaning data**: To run the `scikit-learn` set of models we demonstrate in this notebook, your input dataset must have no missing variables.
3. **Imputing values to missing or irrelevant data**: Once the features are created, always check to make sure the values make sense. You might have some missing values, or impossible values for a given variable (negative values, major outliers). If you have missing values you should think hard about what makes the most sense for your problem; you may want to replace with `0`, the median or mean of your data, or some other value.

4. **Scaling features**: Certain models will have an issue with features on different scales. For example, an individual's age is typically a number between 0 and 100 while earnings can be number between 0 and 1000000 (or higher). In order to circumvent this problem, we can scale our features to the same range (eg

Building a Model and Model Fitting

We need to munge our dataset into our **features** (predictors, or independent variables, or X variables) and **labels** (dependent variables, or Y variables). For ease of reference, in subsequent examples, names of variables that pertain to predictors will start with " `X_` ", and names of variables that pertain to outcome variables will start with " `y_` ".

But it's not enough to just build the model; we're going to need a way to know whether or not it generalizes to new data or can adapt to the future. Convincing others of the quality of results is often the *most challenging* part of an analysis. Building repeatable, well-documented work with clear success metrics makes all the difference.

To convince ourselves - and others - that our modeling results will generalize, we need to hold some data back (not using it to train the model), then apply our model to that hold-out set and "blindly" predict, comparing the model's predictions to what we actually observed. This is called **cross-validation**, and it's the best way we have to estimate how a model will perform on *entirely* novel data. We call the data used to build the model the **training set**, and the rest the **test set**.

In general, we'd like our training set to be as large as possible, to allow our model to be built with as much data as possible. However, you also want to be as confident as possible that your model will generalize to new data. In practice, you'll have to balance these two objectives in a reasonable way.

There are also many ways to split up your data into training and testing sets. Since you're trying to evaluate how your model will perform *in practice*, it's best to emulate the true use case of your model as closely as possible when you decide how to evaluate it. A good tutorial on cross-validation (http://scikit-learn.org/stable/modules/cross_validation.html) can be found on the `scikit-learn` site.

One simple and commonly used method is ***k-fold* cross-validation**, which entails splitting up our dataset into *k* groups, holding out one group while training a model on the rest of the data, evaluating model performance on the held-out "fold," and repeating this process *k* times. Another method is **temporal validation**, which involves building a model using all the data up until a given point in time, and then testing the model on observations that happened after that point.

Our current problem requires us to try to predict an event in the future. Generally, if you use the future to predict the past there will be temporal effects that will help the accuracy of your predictions. We cannot use the future to predict the past in real life, so it is important to use `temporal validation` and create our training and test sets accordingly.

Note: it is important to segregate your data based on time when creating features. Otherwise there can be "leakage," where you accidentally use information that you would not have known at the time. This happens often when calculating aggregation features; for instance, it is quite easy to calculate an average using values that go beyond our training set time-span and not realize it.

Training and Test Sets

We've created two separate prediction cohorts: purchasing 100% whole wheat products in 2015 and 2016. We will use the 2015 cohort as the training set and the 2016 cohort as the test set.

In [ ]:
```
# For the Training Set:
sql = '''
SELECT *
FROM iri_usda_2019_db.ml_model_train
'''

df_training = pd.read_sql(sql, conn)
```

In [ ]:
```
# For the Testing Set:
sql = '''
select *
from iri_usda_2019_db.ml_model_test
'''

df_testing = pd.read_sql(sql, conn)
```

In [ ]:
```
df_training.describe(include='all', percentiles=[.5, .9, .99])
```

In [ ]:
```
df_testing.describe(include='all')
```

Data Distribution

Let's check how much data we have, and what the split is between positive (1) and negative (0) labels in our training dataset. Note that in this case, even though we call it our "positive" outcome, it's actually a bad thing (not buying any 100% whole wheat products). We call it the "positive" outcome because it's what we're trying to identify.

We're going do some basic descriptive statistics with this outcome, because it's good to know what the "baseline" is in our dataset, to be able to intelligently evaluate our performance.

In [ ]:
```
print('Number of rows: {}'.format(df_training.shape[0]))
df_training['label'].value_counts(normalize=True)
```

Crosstabs

Crosstabs can be useful to get a summary of and to find trends and patterns in our data.

In [ ]:
```
pd.crosstab(index=df_training['label'], columns=df_training['wic_june'])
```

In [ ]:
```
pd.crosstab(index=df_testing['label'], columns=df_testing['wic_june'])
```

Selecting Predictors/Features and what we are predicting (Labels)

We'll first list all the columns we have in our data frame and then decide which ones to use as predictors, which one as the label/outcome, and which ones to ignore.

In [ ]:
```
print(list(df_training))
```

In [ ]:
```
print(list(df_testing))
```

In [ ]:
```
# Let's remove the panid from our list of features.
# Testing and training sets should always have the same feature names
sel_features = list(df_training)
sel_features.remove('panid')

# don't want outcome as a feature
sel_features.remove('label')

sel_label = 'label'

#don't want weights as a feature
sel_features.remove('projection61k')
sel_weights = 'projection61k'
```

The code above is aimed at getting a list of columns names for all of the features we are using in our model (`sel_features`), the column name for the label (`sel_label`), and the column name for the survey weights (`sel_weights`).

Checkpoint 1: Anything Else?

Think about any additional features you might want to add to the model. What would be the steps you would need to take to make sure they are included? **Discuss with your group**.

Creating/Formatting Features

Our features are already part of the dataset, but we need to do a little bit of data cleaning and manipulation to get them all into the format we want for our machine learning models.

- All categorical variables must be binary. This means we need to make them into dummy variables. Fortunately, `pandas` has a function to make that easy.
- All numerical variables should be scaled. This doesn't matter for some ML algorithms such as Decision Trees, but it does for others, such as K-Nearest Neighbors.
- Missing values should be accounted for. We need to impute them or drop them, fully understanding how they affect the conclusions we can make.

> Categorical variables that are already binary do not need to be made into dummy variables.

Variables Used in Setup

We created above a list called `sel_features`. This is what we will use to keep track of all of our features to be included in the ML models. When we make dummy variables and scale numerical variables, we'll need to update this list so that it has the most up-to-date features.

Our goal in this formatting step is to end up with:

- `sel_features` which contains a list of all features we will include in our models.
- `train` which contains the training data, with all categorical variables dummied and numerical variables scaled, with column names inside `sel_features`.
- `test` which contains the testing data, with all categorical variables dummied and numerical variables scaled (according to the training data scaling)

Creating dummy variables

Categorical variables need to be converted to a series of binary (0,1) values for `scikit-learn` to use them. We will use the `get_dummies` functions from `pandas` to make our lives easier.

In [ ]:
```
# You can use dtype ... but it may be misleading
df_training['race'].dtype
```

In [ ]:
```
# only want to make object types dummy variables
cols_to_change = ['race', 'hisp', 'ac', 'fed', 'femp', 'med', 'memp', 'm
occ', 'marital',
                  'rentown', 'cats', 'dogs', 'hhtype', 'region', 'wic_ju
ne', 'snap_june', 'hhinc', 'hhsize']
for col in cols_to_change:
    df_training[col] = df_training[col].astype(object)
    df_testing[col] = df_testing[col].astype(object)
```

In [ ]:
```
#confirm columns
df_training.info()
```

In [ ]:
```
# find our "sel_features" columns that are strings
feat_to_dummy = [c for c in sel_features if df_training[c].dtype == 'O']
```

In [ ]:
```
# remove these "feat_to_dummy" columns from our "sel_features" list
for c in feat_to_dummy:
    sel_features.remove(c)
# new "sel_features" list
print(sel_features)
```

In [ ]:
```
# get dummy values
# Note: we are creating a new DataFrame called train and test.
# These are essentially cleaned versions of df_training and df_testing
train = pd.get_dummies(df_training[feat_to_dummy], columns = feat_to_dum
my)
test = pd.get_dummies(df_testing[feat_to_dummy], columns = feat_to_dummy
)
```

In [ ]:
```
# Check to make sure that it created dummies
train.head()
```

In [ ]:
```
# check if column list is the same for train and test dummy sets
sorted(train.columns.tolist()) == sorted(test.columns.tolist())
```

In [ ]:
```
# confirm we have the same dummies for train and test
# if we don't need to fill in for the other
print(train.columns.tolist())
```

In [ ]:
```
#check dummies are the same as train
print(test.columns.tolist())
```

In [ ]:
```
# add dummy columns to full training and testing dataframes
df_training = df_training.merge(train, left_index=True, right_index=True
)
df_testing = df_testing.merge(test, left_index=True, right_index=True)
```

In [ ]:
```
# update the "sel_features" with the dummy columns:
for c in train.columns.tolist():
    sel_features.append(c)
```

In [ ]:
```
# now we can easily access just the columns we want for modeling:
df_training[sel_features].info()
```

Checkpoint 2: Dumb Dummies

What could you do if you have a dummy variable that exists in your testing set but doesn't exist in your training set? Why is this a problem? **Discuss with your group**.

Scaling values

Certain models will have issue with numeric values on different scales. In your analysis cohort, the number of trips taken may vary from ten to 700 while the total expenditures may range from zero to thousands of dollars. Traditional regression methods, for example, tend to result in features (aka right-hand variables, Xs, etc) with small values having larger coefficients than features with large values. On the other hand, some models - like decision trees - are not generally affected by having variables on different scales. To easily use different models, we'll scale all of our continuous data to value between 0 and 1.

We'll start by creating a scaler object using `MinMaxScaler`. We'll `fit` it with the training data and then use it to scale our testing data. This is because we want both the training and testing data to be scaled in the same way. Remember, we're essentially pretending we don't know what's in the testing data for now, so we only scale using the training set, then use that same scaling for any new data (i.e. for the testing data).

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

In [ ]:
```
num_cols = df_training[sel_features].select_dtypes(include = ['float',
'int']).columns
```

In [ ]:
```
# With few features it is relatively easy to hand code which columns to
 scale
# but we can also make our lives a bit easier by doing it programmatical
ly

# get a list columns with values <0 and/or >1:
cols_to_scale = []

for i in num_cols:
    if (df_training[i].min() < 0 or df_training[i].max() > 1):
        cols_to_scale.append(i)
print(cols_to_scale)
```

In [ ]:
```
# add a '*_scl' version of the column for each of our "columns to scale"
# and replace our "sel_features" list

for c in cols_to_scale:
    # create a new column name by adding '_scl' to the end of each colum
n to scale
    new_column_name = c+'_scl'

    # fit MinMaxScaler to training set column
    # reshape because scaler built for 2D arrays
    scaler.fit(df_training[c].values.reshape(-1, 1))

    # update training and testing datasets with new data
    df_training[new_column_name] = scaler.transform(df_training[c].value
s.reshape(-1, 1))
    df_testing[new_column_name] = scaler.transform(df_testing[c].values.
reshape(-1, 1))

    # add new column to our "selection features"
    sel_features.append(new_column_name)

    # and remove the unscaled column
    sel_features.remove(c)
```

In [ ]:
```
sel_features
```

In [ ]:
```
# now our selection features are all scaled between 0-1
df_training[sel_features].describe().T
```

Before running any machine learning algorithms, we have to ensure there are no `NULL` (or `NaN`) values in the data for both our testing and training sets. As you have heard before, **never remove observations with missing values without considering the data you are dropping**. One easy way to check if there are any missing values with `Pandas` is to use the `.info()` method, which returns a count of non-null values for each column in your DataFrame.

In [ ]:
```
df_training[sel_features].info()
```

In [ ]:
```
df_testing[sel_features].info()
```

Here, since we do not have any null values in either dataframe, we do not need to perform any imputation or drop any rows from either the training or testing set.

Model Understanding and Evaluation

In this section, we will run a machine learning model on our training set. The training set's features will be used to predict the labels. Once our model is created using the test set, we will assess its quality by applying it to the test set and by comparing the *predicted values* to the *actual values* for each record in the testing data set.

- **Performance Estimation**: How well will our model do once it is deployed and applied to new data?

Running a Machine Learning Model

Python's `scikit-learn` (http://scikit-learn.org/stable/) is a commonly-used, well-documented Python library for machine learning. This library can help you split your data into training and test sets, fit models and use them to predict results on new data, and evaluate your results.

We will start with the simplest `LogisticRegression` (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) model and see how well that does.

To fit the model, we will create a model object, which we call `logit` here. You can think of this model object as containing all of the instructions necessary to fit that model. Then, we use the `.fit` method in order to give it the data and its corresponding sample weights, and all of the information about the model fit will be contained in `logit`.

In [ ]:
```
# get the underlying numpy.array data for use in scikit-learn
X_train = df_training[sel_features].values
y_train = df_training[sel_label].values
X_test = df_testing[sel_features].values
y_test = df_testing[sel_label].values
```

In [ ]:
```
# Let's fit a model
logit = LogisticRegression(penalty='l1', C=1, solver = 'liblinear')
logit.fit(X_train, y_train, sample_weight = df_training[sel_weights])
print(logit)
```

When we print the model results, we see different parameters we can adjust as we refine the model based on running it against test data (values such as `intercept_scaling`, `max_iters`, `penalty`, and `solver`). Example output:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
      intercept_scaling=1, max_iter=100, multi_class='ovr',
      penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
      verbose=0)
```

To adjust these parameters, one would alter the call that creates the `LogisticRegression()` model instance, passing it one or more of these parameters with a value other than the default. So, to re-fit the model with `max_iter` of 1000, `intercept_scaling` of 2, and `solver` of "lbfgs" (pulled from thin air as an example), you'd create your model as follows:

```
model = LogisticRegression( max_iter = 1000, intercept_scaling = 2, solver =
"lbfgs" )
```

The basic way to choose values for, or "tune," these parameters is the same as the way you choose a model: fit the model to your training data with a variety of parameters, and see which perform the best on the test set. An obvious drawback is that you can also *overfit* to your test set; in this case, you can alter your method of cross-validation.

Model Evaluation

Machine learning models usually do not produce a prediction (0 or 1) directly. Rather, models produce a score between 0 and 1 (that can sometimes be interpreted as a probability), which is basically the model ranking all of the observations from *most likely* to *least likely* to have label of 1. The 0-1 score is then turned into a 0 or 1 based on a threshold.

If you use the sklearn method `.predict()` then the model will select a threshold for you (generally 0.5) - it is almost **never a good idea to let the model choose the threshold for you**. Instead, you should get the actual score and test different threshold values.

In [ ]:
```
# get the prediction scores
y_scores = logit.predict_proba(X_test)[:,1]
```

Look at the distribution of scores:

In [ ]:
```
sns.distplot(y_scores, kde=False, rug=False)
```

In [ ]:
```
df_testing['y_score'] = y_scores
```

In [ ]:
```
# see our selected features and prediction score
df_testing[sel_features + ['y_score']].head()
```

Tools like `sklearn` often have a default threshold of 0.5, but a good threshold is selected based on the data, model and the specific problem you are solving. As a trial run, let's set a threshold to the value of 0.7.

In [ ]:
```
# given the distribution of the scores, what threshold would you set?
selected_threshold = 0.7

# create a list of our predicted outocmes
predicted = y_scores > selected_threshold

# and our actual, or expected, outcomes
expected = y_test
```

Confusion Matrix

Once we have tuned our scores to 0 or 1 for classification, we create a *confusion matrix*, which has four cells: true negatives, true positives, false negatives, and false positives. Each data point belongs in one of these cells, because it has both a ground truth and a predicted label. If an example was predicted to be negative and is negative, it's a true negative. If an example was predicted to be positive and is positive, it's a true positive. If an example was predicted to be negative and is positive, it's a false negative. If an example was predicted to be positive and is negative, it's a false negative.

In [ ]:
```
# Using the confusion_matrix function inside sklearn.metrics
conf_matrix = confusion_matrix(expected,predicted)
print(conf_matrix)
```

The count of true negatives is `conf_matrix[0,0]`, false negatives `conf_matrix[1,0]`, true positives `conf_matrix[1,1]`, and false_positives `conf_matrix[0,1]`.

Accuracy is the ratio of the correct predictions (both positive and negative) to all predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

In [ ]:
```
# generate an accuracy score by comparing expected to predicted.
accuracy = accuracy_score(expected, predicted)
print( "Accuracy = " + str( accuracy ) )
```

In [ ]:
```
df_training['label'].value_counts(normalize=True)
```

Evaluation metrics

What do we think about this accuracy? Good? Bad?

Two metrics that are often more relevant than overall accuracy are **precision** and **recall**.

Precision measures the accuracy of the classifier when it predicts an example to be positive. It is the ratio of correctly predicted positive examples to examples predicted to be positive.

$$Precision = \frac{TP}{TP + FP}$$

Recall measures the accuracy of the classifier to find positive examples in the data.

$$Recall = \frac{TP}{TP + FN}$$

By selecting different thresholds we can vary and tune the precision and recall of a given classifier. A conservative classifier (threshold 0.99) will classify a case as 1 only when it is *very sure*, leading to high precision. On the other end of the spectrum, a low threshold (e.g. 0.01) will lead to higher recall.

In [ ]:
```
# precision_score and recall_score are from sklearn.metrics
precision = precision_score(expected, predicted)
recall = recall_score(expected, predicted)
print( "Precision = " + str( precision ) )
print( "Recall= " + str(recall))
```

If we care about our whole precision-recall space, we can optimize for a metric known as the **area under the curve (AUC-PR)**, which is the area under the precision-recall curve. The maximum AUC-PR is 1.

Checkpoint 3: Evaluation with Different Thresholds

Above, we set the threshold to be 0.7. Try a few different thresholds. What seems like a good threshold value based on precision and recall? Would you like to know any other information before making this decision? **Discuss with your group**.

Plotting the Precision-Recall Curve

In order to see the tradeoff between precision and recall for different thresholds, we can use a visualization that shows both on the same graph. The function `plot_precision_recall` below does this by using the `precision_recall_curve()` function to get the values we want to plot and putting them together. We also print out the AUC for good measure.

In [ ]:
```
def plot_precision_recall(y_true,y_score):
    """
    Plot a precision recall curve

    Parameters
    ----------
    y_true: ls
        ground truth labels
    y_score: ls
        score output from model
    """
    precision_curve, recall_curve, pr_thresholds = precision_recall_curv
e(y_true,y_score)
    plt.plot(recall_curve, precision_curve)
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    auc_val = auc(recall_curve,precision_curve)
    print('AUC-PR: {0:1f}'.format(auc_val))
    plt.show()
    plt.clf()
```

In [ ]:
```
plot_precision_recall(expected, y_scores)
```

Precision and Recall at k%

If we only care about a specific part of the precision-recall curve, we can focus on more fine-grained metrics. For instance, say there is a special program for those most likely to need assistance within the next year, but that it can only cover *1% of our test set*. In that case, we would want to prioritize the 1% who are *most likely* to need assistance within the next year, and it wouldn't matter too much how accurate we were on the overall data.

Let's say that, out of the approximately 300,000 observations, we can intervene on 1% of them, or the "top" 3000 in a year (where "top" means highest likelihood of needing intervention in the next year). We can then focus on optimizing our **precision at 1%**.

In [ ]:
```
def plot_precision_recall_n(y_true, y_prob, model_name):
    """
    y_true: ls
        ls of ground truth labels
    y_prob: ls
        ls of predic proba from model
    model_name: str
        str of model name (e.g, LR_123)
    """
    from sklearn.metrics import precision_recall_curve
    y_score = y_prob
    precision_curve, recall_curve, pr_thresholds = precision_recall_curv
e(y_true, y_score)
    precision_curve = precision_curve[:-1]
    recall_curve = recall_curve[:-1]
    pct_above_per_thresh = []
    number_scored = len(y_score)
    for value in pr_thresholds:
        num_above_thresh = len(y_score[y_score>=value])
        pct_above_thresh = num_above_thresh / float(number_scored)
        pct_above_per_thresh.append(pct_above_thresh)
    pct_above_per_thresh = np.array(pct_above_per_thresh)
    plt.clf()
    fig, ax1 = plt.subplots()
    ax1.plot(pct_above_per_thresh, precision_curve, 'b')
    ax1.set_xlabel('percent of population')
    ax1.set_ylabel('precision', color='b')
    ax1.set_ylim(0,1.05)
    ax2 = ax1.twinx()
    ax2.plot(pct_above_per_thresh, recall_curve, 'r')
    ax2.set_ylabel('recall', color='r')
    ax2.set_ylim(0,1.05)

    name = model_name
    plt.title(name)
    plt.show()
    plt.clf()
```

In [ ]:
```
def precision_at_k(y_true, y_scores,k):

    threshold = np.sort(y_scores)[::-1][int(k*len(y_scores))]
    y_pred = np.asarray([1 if i > threshold else 0 for i in y_scores ])
    return precision_score(y_true, y_pred)
```

In [ ]:
```
plot_precision_recall_n(expected,y_scores, 'LR')
```

In [ ]:
```
p_at_10 = precision_at_k(expected,y_scores, 0.1)
print('Precision at 10%: {:.3f}'.format(p_at_10))
```

Feature Understanding

Now that we have evaluated our model overall, let's look at the coefficients for each feature.

In [ ]:
```
print("The coefficients for each of the features are ")
list(zip(sel_features, logit.coef_[0]))
```

Assessing Model Against Baselines

It is important to check our model against a reasonable **baseline** to know how well our model is doing.

> Without any context, greater than 70% accuracy can sound great... But it's not so great when you remember that oftentimes you could do as well or better by declaring that everyone in the cohort will have a positive/negative outcome, which would be a stupid (not to mention useless) model.

A good place to start is checking against a *random* baseline, assigning every example a label (positive or negative) completely at random. We can use the `random.uniform` function to randomly select 0 or 1, then use that as our "predicted" value to see how well we would have done if we had predicted randomly.

In [ ]:
```
# We will choose to predict on 1% of the population
percent_of_pop = 0.01
```

In [ ]:
```
# Use random.uniform from numpy to generate an array of randomly generat
ed 0 and 1 values of equal length to the test set.
random_score = np.random.uniform(0,1, len(y_test))

# Calculate precision using random predictions
random_p_at_selected = precision_at_k(expected,random_score, percent_of_
pop)
print(random_p_at_selected)
```

Another good practice is checking against an "expert" or rule of thumb baseline.

> This is typically a very simple heuristic. How well would a model of "All WIC participants have a label of 0" perform? You want to make sure your model outperforms these kinds of basic heuristics.

Another good baseline to compare against is the "all label" (label is always 1). Our "model" in this case is that we always predict 1 and see how our measures perform with that.

In [ ]:
```
none_predicted = np.array([1 for i in range(df_testing.shape[0])])
none_precision = precision_score(expected, none_predicted)
print(none_precision)
```

In [ ]:
```
model_precision = precision_at_k(expected, y_scores, percent_of_pop)
```

In [ ]:
```
sns.set_style("white")
sns.set_context("poster", font_scale=1.25, rc={"lines.linewidth":1.25,
"lines.markersize":4})
fig, ax = plt.subplots(1, figsize=(10,8))
sns.barplot(['Random','None Employed', 'Our Model'],
#             [random_p_at_1, none_precision, expert_precision, max_p_at
_k],
            [random_p_at_selected, none_precision, model_precision],
#             palette=['#6F777D','#6F777D','#6F777D','#800000'])
            palette=['#6F777D','#6F777D','#800000'])
sns.despine()
plt.ylim(0,1)
plt.ylabel('precision at {}%'.format(percent_of_pop*100));
```

Checkpoint 4: Running another model

Let's try running a different model, using decision trees this time. The `sklearn` package actually makes it quite easy to run alternative models. All you need to do is create that model object, then use the `fit` method using your training data, and you're all set! That is, we can use the code below:

In [ ]:
```
# packages to display a tree in Jupyter notebooks
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import graphviz as gv
import pydotplus
```

In [ ]:
```
tree = DecisionTreeClassifier(max_depth = 3)
tree.fit(X_train, y_train)
```

Just like we fit our logistic regression model above and used that model object to get predicted scores, we can do the same with this Decision Tree Classifier model object. That is, we can get our predicted scores using this model using the following code:

In [ ]:
```
tree_predicted = tree.predict_proba(X_test)[:,0]
tree_predicted
```

In [ ]:
```
# visualize the tree

# object to hold the graphviz data
dot_data = StringIO()

# create the visualization
export_graphviz(tree, out_file=dot_data, filled=True,
                rounded=True, special_characters=True,
                feature_names=df_training[sel_features].columns.values)

# convert to a graph from the data
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

# you can print out the graph to a pdf with this line:
# graph.write_pdf('./output/model_eval_tree1.pdf')

# or view it directly in notebook
Image(graph.create_png())
```

Using the predicted scores from the tree model, try calculating the precision and recall for varying values of the threshold. Use the functions created above to create precision-recall graphs. Do you think the tree model is performing better or worse than the logistic regression model?

Machine Learning Pipeline

When working on machine learning projects, it is a good idea to structure your code as a modular **pipeline**, which contains all of the steps of your analysis, from the original data source to the results that you report, along with documentation. This has many advantages:

- **Reproducibility**. It's important that your work be reproducible. This means that someone else should be able to see what you did, follow the exact same process, and come up with the exact same results. It also means that someone else can follow the steps you took and see what decisions you made, whether that person is a collaborator, a reviewer for a journal, or the agency you are working with.
- **Ease of model evaluation and comparison**.
- **Ability to make changes.** If you receive new data and want to go through the process again, or if there are updates to the data you used, you can easily substitute new data and reproduce the process without starting from scratch.

Survey of Algorithms

We have only scratched the surface of what we can do with our model. We've only tried two classifiers (Logistic Regression and a Decision Tree), and there are plenty more classification algorithms in `sklearn`. Let's try them!

In [ ]:
```
clfs = {'RF': RandomForestClassifier(n_estimators=1000, n_jobs=-1),
        'ET': ExtraTreesClassifier(n_estimators=1000, n_jobs=-1),
        'LR': LogisticRegression(penalty='l1', C=1e5),
        'SGD':SGDClassifier(loss='log'),
        'GB': GradientBoostingClassifier(learning_rate=0.05, subsample=
0.5, max_depth=6, random_state=17
                                         , n_estimators=10),
        'NB': GaussianNB(),
        'DT': DecisionTreeClassifier(max_depth=10, min_samples_split=10)
       }
```

In [ ]:
```
sel_clfs = ['RF', 'ET', 'LR', 'SGD', 'GB', 'NB', 'DT']
```

In [ ]:
```
max_p_at_k = 0
df_results = pd.DataFrame()
for clfNM in sel_clfs:
    clf = clfs[clfNM]
    clf.fit( X_train, y_train )
#     print(clf)
    y_score = clf.predict_proba(X_test)[:,1]
    predicted = np.array(y_score)
    expected = np.array(y_test)
    plot_precision_recall_n(expected,predicted, clfNM)
    p_at_1 = precision_at_k(expected,y_score, 0.01)
    p_at_5 = precision_at_k(expected,y_score,0.05)
    p_at_10 = precision_at_k(expected,y_score,0.10)
    p_at_30 = precision_at_k(expected,y_score,0.30)
    fpr, tpr, thresholds = roc_curve(expected,y_score)
    auc_val = auc(fpr,tpr)
    df_results = df_results.append([{
        'clfNM':clfNM,
        'p_at_1':p_at_1,
        'p_at_5':p_at_5,
        'p_at_10':p_at_10,
        'auc':auc_val,
        'clf': clf
    }])

    #feature importances
    if hasattr(clf, 'coef_'):
        feature_import = dict(
            zip(sel_features, clf.coef_.ravel()))
    elif hasattr(clf, 'feature_importances_'):
        feature_import = dict(
            zip(sel_features, clf.feature_importances_))
    print("FEATURE IMPORTANCES")
    print(feature_import)

    if max_p_at_k < p_at_1:
        max_p_at_k = p_at_1
    print('Precision at 1%: {:.2f}'.format(p_at_1))
# df_results.to_csv('output/modelrun.csv')f
```

Let's view the best model at 1%

In [ ]:
```
sns.set_style("white")
sns.set_context("poster", font_scale=1.25, rc={"lines.linewidth":1.25,
"lines.markersize":8})
fig, ax = plt.subplots(1, figsize=(10,6))
sns.barplot(['Random','None Employed', 'Best Model'],
#             [random_p_at_1, none_precision, expert_precision, max_p_at
_k],
            [random_p_at_selected, none_precision, max_p_at_k],
#             palette=['#6F777D','#6F777D','#6F777D','#800000'])
            palette=['#6F777D','#6F777D','#800000'])
sns.despine()
plt.ylim(0,1)
plt.ylabel('precision at 1%')
```

In [ ]:
```
# view all saved evaluation metrics
df_results
```

Exercise

Our model has just scratched the surface. Try the following:

- Create more features
- Try more models
- Try different parameters for your model

The notebook used to create features and labels is the Data Preparation (04_01_ML_Data_Prep.ipynb) notebook. Take the time to look at it and understand how every metric was created and added to the data table.

Additional Resources

- Hastie et al.'s The Elements of Statistical Learning (http://statweb.stanford.edu/~tibs/ElemStatLearn/) is a classic and is available online for free.
- James et al.'s An Introduction to Statistical Learning (http://www-bcf.usc.edu/~gareth/ISL/), also available online, includes less mathematics and is more approachable.
- Wu et al.'s Top 10 Algorithms in Data Mining (http://www.cs.uvm.edu/~icdm/algorithms/10Algorithms-08.pdf).