```
In [200]: %matplotlib inline
          import matplotlib
          import seaborn as sns
          sns.set()
          matplotlib.rcParams['figure.dpi'] = 144
```

```
In [201]: from static_grader import grader
```

Part 1: Regression

We'll start by looking at some demographic and socioeconomic data (https://catalog.data.gov/dataset/rural-veterans-by-state-2015) from the Department of Veterans Affairs. Our goal is to build a linear model that predicts an individual's household income. Let's load in the data:

```
In [202]: !aws s3 sync s3://dataincubator-course/mldata/ . --exclude '*' --include 'vet_
          data.csv'
```

Scikit-Learn is pretty flexible when it comes to data formats, so there is no right way to import the data. Using Pandas (https://pandas.pydata.org/) DataFrames makes it easier to keep track of feature names, although you may use NumPy arrays or even Python lists.

```
In [203]: import numpy as np
          import pandas as pd

          vet_data = pd.read_csv("./vet_data.csv", index_col=0)

          print(type(vet_data))
          vet_data.shape

          <class 'pandas.core.frame.DataFrame'>

Out[203]: (20000, 16)
```

You can get a nice preview of the data in a DataFrame with its `.head()` method.

In [204]:
```
vet_data.head()
```

Out[204]:

	HINCP	AGEP	vet	period	HICOV	SEX	pregion	racepct	marital	school	emppct
0	175000.0	83	2	3.0	1	2	3	1	3	2	3.0
1	54000.0	67	1	2.0	1	1	4	1	4	2	3.0
2	84000.0	53	2	3.0	1	2	3	1	1	2	1.0
3	49700.0	61	2	3.0	1	1	4	4	1	2	3.0
4	40000.0	73	1	3.0	1	1	2	1	3	2	3.0

Column descriptions can be found in this slightly unwieldy data dictionary (https://www.data.va.gov/sites/default/files/Data-Dictionary-Apps-for-Ag.pdf).

Age Model

We'll start with the first two columns in the data set. `HINCP` is household income in dollars, and `AGEP` is each individual's age in years. Use Scikit-Learn's `LinearRegression` (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html) to make a simple model that predicts income based on age.

In this case our feature matrix `X` and our label vector `y` are just the `HINCP` and `AGEP` columns from the data set. Note that Scikit-learn expects `X` to be a 2-dimensional `nd-array` or a DataFrame and `y` to be a 1-dimensional `nd-array` or a DataSeries, so we'll select these columns in slightly different ways. We'll also convert `X` and `y` to NumPy arrays to match the format of the test set used by the grader. This isn't really necessary, but it may save you some grief if you want to write a custom estimator later on.

In [205]:
```
from sklearn.linear_model import LinearRegression

X = vet_data[['AGEP']] # List of columns gives a DataFrame
y = vet_data['HINCP']  # Single column gives a DataSeries

X, y = X.values, y.values # Convert to NumPy arrays
```

Now create and fit a `LinearRegression` object, and pass its `predict` method to the grader for scoring.

In [206]:
```
age_est = LinearRegression()
age_est.fit(X, y)
```

Out[206]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

In [207]:
```
grader.score('intro_ml__age_model', age_est.predict)
```
```
==================
Your score:  1.0
==================
```

Regressors in Scikit-learn have a `.score` method which returns an R^2 score. Let's see how well we did.

In [208]:
```
age_est.score(X, y)
```
Out[208]:
```
0.004548325066588199
```

Here the low R^2 score indicates underfitting. In other words, our model isn't very good. This isn't surprising, since we've ignored most of our data. Furthermore, age and income may not be linearly correlated (try plotting them)! We'll get better results by including more features and using more sophisticated models.

Full Linear Model

Next, we'll build a linear model that uses all of the features in the data set instead of just one.

In [209]:
```
X = vet_data.drop('HINCP', axis=1).values
y = vet_data['HINCP'].values
```

This might seem like a repeat of the previous question, but there's a problem: all of the other features in our data set are categorical!

For example, the `marital` column represents marital status using the following codes:

> 1: married
>
> 2: divorced
>
> 3: widowed, separated
>
> 4: never married

The numerical order of these labels isn't meaningful (someone who's divorced isn't more married than someone who's separated), so it doesn't make sense to feed them directly into a linear model. A better alternative is to use one hot encoding, effectively creating a new indicator variable for each label.

We can do this using Scikit-Learn's `OneHotEncoder` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html).

In [210]:
```
from sklearn.preprocessing import OneHotEncoder
```

As an example, suppose we have a matrix with two categorical features. The first feature takes on the values {1,2,3,4} while the second takes on the values {10,20}.

```
In [211]: cat_feats = np.array([[1,10],[2,20],[3,10],[4,20],[3,10],[2,20],[1,10]])
          cat_feats
```

```
Out[211]: array([[ 1, 10],
                 [ 2, 20],
                 [ 3, 10],
                 [ 4, 20],
                 [ 3, 10],
                 [ 2, 20],
                 [ 1, 10]])
```

Now let's run this through a one-hot encoder.

```
In [212]: OneHotEncoder(sparse=False).fit_transform(cat_feats)
```

```
Out[212]: array([[1., 0., 0., 0., 1., 0.],
                 [0., 1., 0., 0., 0., 1.],
                 [0., 0., 1., 0., 1., 0.],
                 [0., 0., 0., 1., 0., 1.],
                 [0., 0., 1., 0., 1., 0.],
                 [0., 1., 0., 0., 0., 1.],
                 [1., 0., 0., 0., 1., 0.]])
```

Note that the first four columns encode four values taken by the first feature, while the final two columns of the output represent the two values taken by the second feature.

Question: Why is it necessary to fit the `OneHotEncoder`?

Create a transformer to do one hot encoding on our feature matrix `X`.

We don't want to encode the `AGEP` column since that feature is already continuous, so you'll need to put a mask into the `categorical_features` argument of `OneHotEncoder` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html). You should also be aware of the `sparse` argument, which controls whether the output is a sparse matrix or a NumPy array. Sparse output is more memory efficient, and it's the default, but other functions may not accept it as input.

```
In [213]: one_hot_transformer = OneHotEncoder(sparse=False, categorical_features=[1, 2,
          3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Now we could apply this transformer to our data using its `.fit_transform` method, but applying transformers directly generally isn't good practice:

- Storing the original data set and the transformed data set as two different objects clutters our workspace. There's a non-trivial chance that we'll confuse one for the other.
- We'll need to apply the transformer again each time we encounter new data (for example, to make predictions). This is an extra step that we might forget, and it also means that we're repeating code.
- Both of these problems are compounded when we want to apply many transformers in sequence.

Instead we should use Scikit-Learn's `Pipeline` (http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) function to organize our transformers and estimator into a single object which only needs to be applied to the data once.

Try building a pipeline that combines our `one_hot_transformer` with a `LinearRegression()` estimator.

```
In [214]: from sklearn.pipeline import Pipeline

          linear_est = Pipeline([("one hot transformer", one_hot_transformer),
                                  ("linear regression", LinearRegression())
                                ])

          linear_est.fit(X,y)

Out[214]: Pipeline(memory=None,
               steps=[('one hot transformer', OneHotEncoder(categorical_features=[1, 2,
          3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14],
                 dtype=<class 'numpy.float64'>, handle_unknown='error',
                 n_values='auto', sparse=False)), ('linear regression', LinearRegressio
          n(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False))])
```

```
In [215]: grader.score('intro_ml__linear_model', linear_est.predict)

          ==================
          Your score:  1.0
          ==================
```

Polynomial Features

Linear models cannot detect interactions between features. One way around this limitation is to create new features that encode the interactions we're interested in. For example, we can use the values given by the product of each pair (or tuple) of features. This is exactly what Scikit-Learn's `PolynomialFeatures` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html) transformer does.

```
In [216]: from sklearn.preprocessing import PolynomialFeatures

          polynomial_transformer = PolynomialFeatures(degree=2, interaction_only=True, i
          nclude_bias=False)

          # For example, if our original features are x,y,z, then the new features are x
          y,xz,yz
          polynomial_transformer.fit_transform([[2,3,5]])

Out[216]: array([[ 2.,  3.,  5.,  6., 10., 15.]])
```

If we apply `one_hot_transformer` and `polynomial_transformer` in sequence, then we'll have more features to build a linear model on top of. Try applying these transformers to the test data given below.

- We want to make sure that the columns of the output have the same meaning as when we apply these transformers to our training set `X`, so think about the difference between `.transform` and `.fit_transform` as you apply each transformer.
- Make sure that you treated `AGEP` as a continuous variable when you defined `one_hot_transformer`.

```
In [217]: test_data = [[ 31.,   2.,   3.,   1.,   2.,   2.,   1.,   1.,   2.,   3.,   0.
          ,   2.,   2.,   1.,   1.],
                       [ 20.,   2.,   3.,   1.,   1.,   3.,   1.,   4.,   2.,   1.,   0.
          ,   2.,   1.,   2.,   1.],
                       [ 22.,   2.,   3.,   1.,   2.,   4.,   1.,   1.,   1.,   3.,   1.
          ,   2.,   2.,   1.,   1.],
                       [ 73.,   1.,   3.,   1.,   1.,   1.,   1.,   1.,   2.,   3.,   0.
          ,   1.,   1.,   1.,   1.],
                       [ 68.,   2.,   3.,   1.,   2.,   1.,   1.,   3.,   2.,   3.,   0.
          ,   1.,   1.,   1.,   1.]]
```

```
In [ ]: transformed_test_data = ...
```

Your answer should have 5 rows and 1128 columns.

```
In [ ]: assert(transformed_test_data.shape == (5,1128))
```

```
In [ ]: grader.score('intro_ml__trans_data', lambda: transformed_test_data)
```

If you look at the output, you may notice that most of the entries are zero!

```
In [ ]: # The percentage of nonzero elements in our output.
        100*(transformed_test_data!=0).sum()/(5.*1128)
```

Unfortunately, this is an inevitable consequence of using `OneHotEncoder` and `PolynomialFeatures` together. Most of the variables coming from one hot encoding are equal to zero most of the time, and it follows that the cross terms will equal zero with an even greater frequency. For this reason, `PolynomialFeatures` is usually used with features which are continuous to begin with.

This doesn't mean that our transformed data is useless, but variables that are zero most of the time aren't contributing as much to our model, and giving them too much weight may contribute to overfitting. In our case, a better approach is to switch to a non-linear model.

Decision Tree Regressor

A decision tree is a binary tree that sorts data into groups by comparing observations to reference values one feature at a time. Different features can be considered at different nodes, so decision trees are capable of encoding non-linear behavior, effectively capturing interactions between features.

Decision trees can also make a sequence of splits based on a single feature, effectively approximating non-linear functions of a single variable. This is especially relevant because of the `AGEP` variable that we discussed above. Intuition should tell us that the dependence of income on age is not linear. So we'll have better luck with a non-linear model.

Create a new model using `DecisionTreeRegressor` (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html).

In [ ]:
```
from sklearn.tree import DecisionTreeRegressor

tree_est = ...
```

Some things to consider:

- By default, a decision tree will keep growing until it has pure leaves. For regression, this can mean that we get a different leaf for each sample. Obviously this is a recipe for overfitting, so we should limit the growth of the tree by setting hyperparameters like `max_depth` or `min_samples_leaf`. `GridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) can help you find the optimal values.
- Is one hot encoding still necessary? Why or why not?

In [ ]:
```
grader.score('intro_ml__tree_model', tree_est.predict)
```

Part 2: Classification

Next we'll look at delay and cancellation data for airline flights from 2008. We'll only consider flights from two airline carriers: Southwest Airlines and American Airlines. Our goal is to build a classifier that predicts which carrier each flight belongs to.

Let's start by loading in the data.

In [ ]:
```
!aws s3 sync s3://dataincubator-course/mldata/ . --exclude '*' --include 'ml_f
light_data.csv'
```

In [ ]:
```
flight_data = pd.read_csv("./ml_flight_data.csv", index_col=0)

print(type(flight_data), flight_data.shape)

flight_data.head()
```

A brief description of the columns can be found here (http://stat-computing.org/dataexpo/2009/the-data.html). For convenience, we've converted times from `hhmm` format to hours and dropped some columns. All features can be treated as continuous.

To start, we'll select the appropriate columns to get our feature and label sets, `X` and `y`. Depending on your strategy for model selection and evaluation, you may wish to further split these into test and training sets using Scikit-Learn's `train_test_split` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html).

In [ ]:
```
X = flight_data.drop('Carrier', axis=1).values
y = flight_data['Carrier'].values
```

Logistic Regression Model

We'll start by building a classifier that uses Logistic Regression. Practically speaking, this could be very similar to the models that we built before, but we have a new challenge to deal with: our data has missing values.

In a different situation, we might want to drop rows or columns, but this often means throwing out good data, and we may not have the luxury of ignoring incomplete observations when the time comes to apply our model. Here we'll assign a value to each missing field based on the other values in the same column. We suggest using the `Imputer` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Imputer.html) transformer from Scikit-Learn's preprocessing module.

It should be noted that the behavior of `Imputer` is somewhat limited. If we wanted to use a more complicated strategy to impute values (or to use different strategies for different columns), then we'd need to write a custom transformer. Another strategy to be aware of is filling in random values. This is a middle ground between imputation and dropping rows/columns.

In [ ]:
```
from sklearn.preprocessing import Imputer
from sklearn.linear_model import LogisticRegression

log_est = ...
```

Once you have a working pipeline, try modifying it to improve performance.

- Consider the `StandardScaler` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) transformer. Is it appropriate to use it here?
- You can use `GridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) to do hyperparameter selection. `LogisticRegression` has hyperparameter, `C`, which controls regularization, and `Imputer` has an argument, `strategy`, that controls how imputed values are calculated.

In [ ]:
```
grader.score('intro_ml__log_model', log_est.predict)
```

Feature Importance

One way to gain insight into our machine learning models is to look at how much influence each variable has on their predictions. In a broad sense, variables with more influence are more important because they have more predictive power. For example, with linear or logistic regression we can measure importance by looking at the coefficients learned by the model. If our features were normalized to begin with, then the coefficients with the greatest absolute value correspond to the most predictive features.

In this question, we'll use the `.coef_` attribute of `LogisticRegression` to get the coefficients from the estimator we built in the previous step. Keep in mind that we need to do feature scaling to get a fair comparison of the coefficients, so you may need to modify your pipeline to include `StandardScaler`.

Then write a function that returns a list of the five most important features, together with their coefficients. Depending on how you built your estimator, you may need to do some digging to access the `LogisticRegression` component. The `best_estimator_` attribute of `GridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) and the `named_steps` attribute of `Pipeline` (http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) may come in handy.

In [ ]:
```
column_names = list(flight_data.drop('Carrier', axis = 1))

top_5 = [('Distance', 0.421220771945)] * 5

grader.score('intro_ml__feature_importances', top_5)
```