Chapter 1

1)

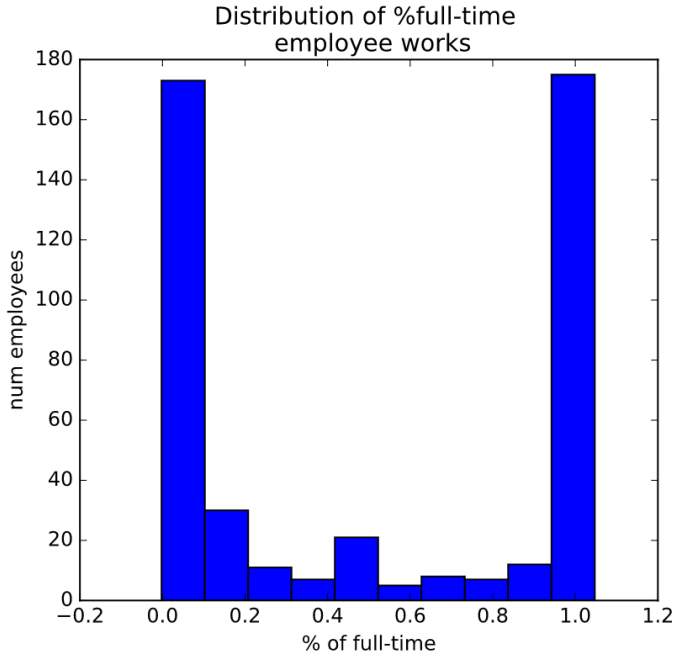
```
# Print the summary statistics
print(df.describe())
```

```
              FTE         Total
count  449.000000  1.542000e+03
mean     0.493532  1.446867e+04
std      0.452844  7.916752e+04
min     -0.002369 -1.044084e+06
25%           NaN           NaN
50%           NaN           NaN
75%           NaN           NaN
max      1.047222  1.367500e+06
```

```
# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
å
# Create the histogram
plt.hist(df['FTE'].dropna())

# Add title and labels
plt.title('Distribution of %full-time \n employee works')
plt.xlabel('% of full-time')
plt.ylabel('num employees')

# Display the histogram
plt.show()
```



Note)

Encode labels as categories

- ML algorithms work on numbers, not strings
  - Need a numeric representation of these strings
- Strings can be slow compared to numbers
- In pandas, 'category' dtype encodes categorical data numerically
  - Can speed up code

Encode labels as categories (sample

```
In [1]: sample_df.label.head(2)
Out[1]:
0    a
1    b
Name: label, dtype: object

In [2]: sample_df.label = sample_df.label.astype('category')

In [3]: sample_df.label.head(2)
Out[3]:
0    a
1    b
Name: label, dtype: category
Categories (2, object): [a, b]
```

Dummy variable encoding

```
In [4]: dummies = pd.get_dummies(sample_df[['label']], prefix_sep='_')

In [5]: dummies.head(2)
Out[5]:
   label_a  label_b
0        1        0
1        0        1
```

- Also called a 'binary indicator' representation

Encode labels as categories

```
In [7]: categorize_label = lambda x: x.astype('category')

In [8]: sample_df.label = sample_df[['label']].apply(categorize_label,
   ...:                   axis=0)

In [9]: sample_df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 0 to 99
Data columns (total 4 columns):
label           100 non-null category
numeric         100 non-null float64
text            100 non-null object
with_missing    95 non-null float64
dtypes: category(1), float64(2), object(1)
memory usage: 3.2+ KB
```

2)

df.dtypes.value_counts()

object 23
float64 2
dtype: int64

3)

# Define the lambda function: categorize_label
print(df[LABELS].dtypes)

Function object
Use object
Sharing object
Reporting object
Student_Type object
Position_Type object
Object_Type object
Pre_K object
Operating_Status object
dtype: object

categorize_label = lambda x: x.astype('category')

# Convert df[LABELS] to a categorical type
df[LABELS] = df[LABELS].apply(categorize_label,axis=0)

# Print the converted dtypes
print(df[LABELS].dtypes)

```
Function          category
Use               category
Sharing           category
Reporting         category
Student_Type      category
Position_Type     category
Object_Type       category
Pre_K             category
Operating_Status  category
dtype: object
```

4)

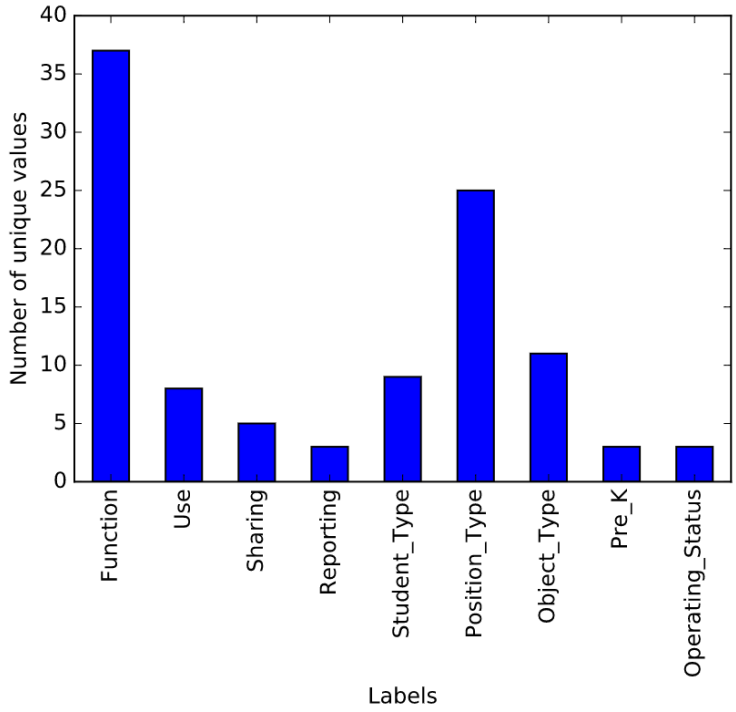
```
# Import matplotlib.pyplot
import matplotlib.pyplot as plt

# Calculate number of unique values for each label: num_unique_labels
num_unique_labels = df[LABELS].apply(pd.Series.nunique)

# Plot number of unique values for each label
num_unique_labels.plot(kind='bar')

# Label the axes
plt.xlabel('Labels')
plt.ylabel('Number of unique values')

# Display the plot
plt.show()
```



Note)

Computing log loss with NumPy

logloss.py

```
import numpy as np

def compute_log_loss(predicted, actual, eps=1e-14):
    """ Computes the logarithmic loss between predicted and
        actual when these are 1D arrays.

        :param predicted: The predicted probabilities as floats between 0-1
        :param actual: The actual binary labels. Either 0 or 1.
        :param eps (optional): log(0) is inf, so we need to offset our
                               predicted values slightly by eps from 0 or 1.
    """
    predicted = np.clip(predicted, eps, 1 - eps)
    loss = -1 * np.mean(actual * np.log(predicted)
              + (1 - actual)
              * np.log(1 - predicted))

    return loss
```

5)

```
# Compute and print log loss for 1st case
correct_confident = compute_log_loss(correct_confident, actual_labels)
print("Log loss, correct and confident: {}".format(correct_confident))

# Compute log loss for 2nd case
correct_not_confident = compute_log_loss(correct_not_confident,
actual_labels)
print("Log loss, correct and not confident: {}".format(correct_not_confident))

# Compute and print log loss for 3rd case
wrong_not_confident = compute_log_loss(wrong_not_confident, actual_labels)
print("Log loss, wrong and not confident: {}".format(wrong_not_confident))

# Compute and print log loss for 4th case
wrong_confident = compute_log_loss(wrong_confident,actual_labels)
print("Log loss, wrong and confident: {}".format(wrong_confident))

# Compute and print log loss for actual labels
actual_labels = compute_log_loss(actual_labels, actual_labels)
print("Log loss, actual labels: {}".format(actual_labels))
```

```
Log loss, correct and confident: 0.05129329438755058
Log loss, correct and not confident: 0.4307829160924542
Log loss, wrong and not confident: 1.049822124498678
Log loss, wrong and confident: 2.9957322735539904
Log loss, actual labels: 9.99200722162646e-15
```

Chapter 2

Splitting the data

```
In  [1]: data_to_train = df[NUMERIC_COLUMNS].fillna(-1000)

In  [2]: labels_to_use = pd.get_dummies(df[LABELS])

In  [3]: X_train, X_test, y_train, y_test = multilabel_train_test_split(
                                             data_to_train, labels_to_use,
                                             size=0.2, seed=123)
```

1)

The first step is to split the data into a training set and a test set. Some labels don't occur very often, but we want to make sure that they appear in both the training and the test sets. We provide a function that will make sure at least `min_count` examples of each label appear in each split: `multilabel_train_test_split`.

```
# Create the new DataFrame: numeric_data_only
numeric_data_only = df[NUMERIC_COLUMNS].fillna(-1000)

# Get labels and convert to dummy variables: label_dummies
label_dummies =pd.get_dummies(df[LABELS])

# Create training and test sets
X_train, X_test, y_train, y_test =
multilabel_train_test_split(numeric_data_only,
                                                label_dummies,
                                                size=0.2,
                                                seed=123)

# Print the info
print("X_train info:")
print(X_train.info())
```

```
X_train info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1040 entries, 198 to 101861
Data columns (total 2 columns):
FTE      1040 non-null float64
Total    1040 non-null float64
dtypes: float64(2)
memory usage: 24.4 KB
```

```
print("\nX_test info:")
```

```
print(X_test.info())
```
```
X_test info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 520 entries, 209 to 448628
Data columns (total 2 columns):
FTE      520 non-null float64
Total    520 non-null float64
dtypes: float64(2)
memory usage: 12.2 KB
```
```
print("\ny_train info:")
print(y_train.info())
```
```
y_train info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1040 entries, 198 to 101861
Columns: 104 entries, Function_Aides Compensation to Operating_Status_PreK-12 Operating
dtypes: float64(104)
memory usage: 853.1 KB
```
```
print("\ny_test info:")
print(y_test.info())
```
```
y_test info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 520 entries, 209 to 448628
Columns: 104 entries, Function_Aides Compensation to Operating_Status_PreK-12 Operating
dtypes: float64(104)
memory usage: 426.6 KB
```

2)

```
# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier


# Create the DataFrame: numeric_data_only
numeric_data_only = df[NUMERIC_COLUMNS].fillna(-1000)

# Get labels and convert to dummy variables: label_dummies
label_dummies = pd.get_dummies(df[LABELS])

# Create training and test sets
X_train, X_test, y_train, y_test =
multilabel_train_test_split(numeric_data_only,
                                                label_dummies,
                                                size=0.2,
                                                seed=123)
```

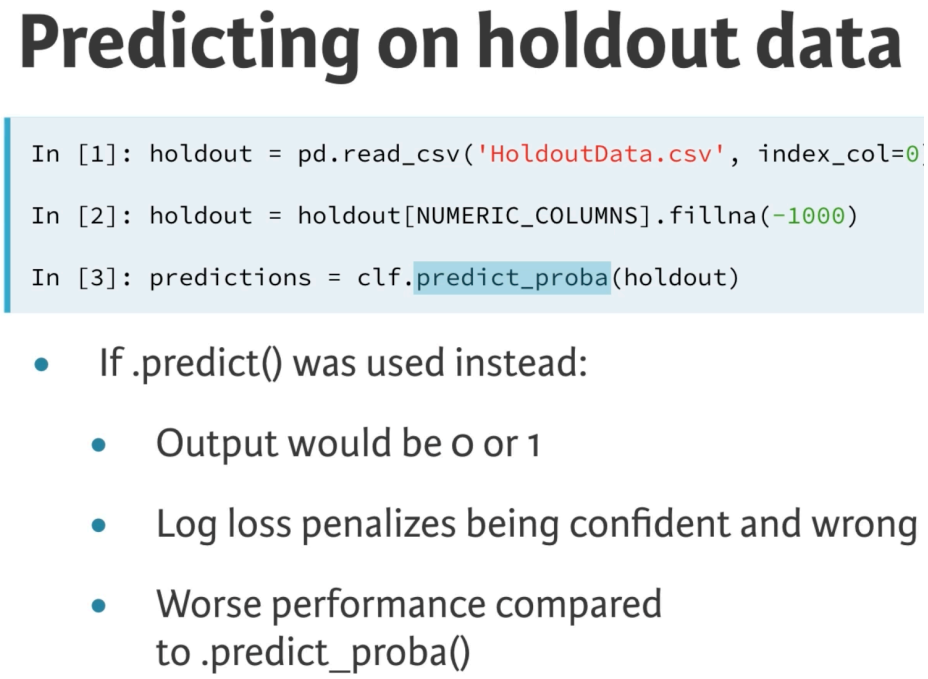
```
# Instantiate the classifier: clf
clf = OneVsRestClassifier(LogisticRegression())

# Fit the classifier to the training data
clf.fit(X_train,y_train)

# Print the accuracy
print("Accuracy: {}".format(clf.score(X_test, y_test)))
```

Accuracy: 0.0

Note)



3)

```
# Instantiate the classifier: clf
clf = OneVsRestClassifier(LogisticRegression())

# Fit it to the training data
clf.fit(X_train, y_train)

# Load the holdout data: holdout
holdout = pd.read_csv('HoldoutData.csv',index_col=0)

# Generate predictions: predictions
predictions = clf.predict_proba(holdout[NUMERIC_COLUMNS].fillna(-1000))
```

4)

```
# Generate predictions: predictions
predictions = clf.predict_proba(holdout[NUMERIC_COLUMNS].fillna(-1000))

# Format predictions in DataFrame: prediction_df
prediction_df =
pd.DataFrame(columns=pd.get_dummies(df[LABELS]).columns,
                        index=holdout.index,
                        data=predictions)


# Save prediction_df to csv
prediction_df.to_csv('predictions.csv')

# Submit the predictions for scoring: score
score = score_submission(pred_path='predictions.csv')

# Print score
print('Your model, trained with numeric data only, yields logloss score:
{}'.format(score))
```

Your model, trained with numeric data only, yields logloss score: 1.9067227623381413

Note)

Using CountVectorizer() on column of main dataset

```
In [1]: from sklearn.feature_extraction.text import CountVectorizer

In [2]: TOKENS_BASIC = '\\S+(?=\\s+)'

In [3]: df.Program_Description.fillna('', inplace=True)

In [4]: vec_basic = CountVectorizer(token_pattern=TOKENS_BASIC)
```

5)

- Fill missing values in `df.Position_Extra` using `.fillna('')` to replace NaNs with empty strings. Specify the additional keyword argument `inplace=True` **so that you don't have to assign the result back to** `df`.

```
# Import CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'
```

```
# Fill missing values in df.Position_Extra
df.Position_Extra.fillna('',inplace=True)

# Instantiate the CountVectorizer: vec_alphanumeric
vec_alphanumeric =
CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Fit to the data
vec_alphanumeric.fit(df.Position_Extra)

# Print the number of tokens and first 15 tokens
msg = "There are {} tokens in Position_Extra if we split on non-alpha numeric"
print(msg.format(len(vec_alphanumeric.get_feature_names())))
print(vec_alphanumeric.get_feature_names()[:15])
```

```
There are 123 tokens in Position_Extra if we split on non-alpha numeric
['1st', '2nd', '3rd', 'a', 'ab', 'additional', 'adm', 'administrative', 'and', 'any', 'art', 'assessment', 'assistant',
'asst', 'athletic']
```

6)
Combining text columns for tokenization

In order to get a bag-of-words representation for all of the text data in our DataFrame, you must first convert the text data in each row of the DataFrame into a single string.
In the previous exercise, this wasn't necessary because you only looked at one column of data, so each row was already just a single string. `CountVectorizer` expects each row to just be a single string, so in order to use all of the text columns, you'll need a method to turn a list of strings into a single string.
In this exercise, you'll complete the function definition `combine_text_columns()`. When completed, this function will convert all training text data in your DataFrame to a single string per row that can be passed to the vectorizer object and made into a bag-of-words using the `.fit_transform()` method.
Note that the function uses `NUMERIC_COLUMNS` and `LABELS` to determine which columns to drop. These lists have been loaded into the workspace.

```
# Define combine_text_columns()
def combine_text_columns(data_frame, to_drop=NUMERIC_COLUMNS +
LABELS):
    """ converts all text in each row of data_frame to single vector """
```

```
    # Drop non-text columns that are in the df
    to_drop = set(to_drop) & set(data_frame.columns.tolist())
    text_data = data_frame.drop(to_drop,axis=1)

    # Replace nans with blanks
    text_data.fillna('',inplace=True)

    # Join all text items in a row that have a space in between
    return text_data.apply(lambda x: " ".join(x), axis=1)
```

7)

```
# Import the CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

# Create the basic token pattern
TOKENS_BASIC = '\\S+(?=\\s+)'

# Create the alphanumeric token pattern
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate basic CountVectorizer: vec_basic
vec_basic = CountVectorizer(token_pattern=TOKENS_BASIC)

# Instantiate alphanumeric CountVectorizer: vec_alphanumeric
vec_alphanumeric =
CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Create the text vector
text_vector = combine_text_columns(df)

# Fit and transform vec_basic
vec_basic.fit_transform(text_vector)

# Print number of tokens of vec_basic
print("There are {} tokens in the
dataset".format(len(vec_basic.get_feature_names())))
```

There are 1405 tokens in the dataset

```
# Fit and transform vec_alphanumeric
vec_alphanumeric.fit_transform(text_vector)

# Print number of tokens of vec_alphanumeric
```

```
print("There are {} alpha-numeric tokens in the
dataset".format(len(vec_alphanumeric.get_feature_names())))
```
There are 1117 alpha-numeric tokens in the dataset

8)

```
# Import Pipeline
from sklearn.pipeline import Pipeline

# Import other necessary modules
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

# Split and select numeric data only, no nans
X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric']],
                                    pd.get_dummies(sample_df['label']),
                                    random_state=22)

# Instantiate Pipeline object: pl
pl = Pipeline([
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

# Fit the pipeline to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - numeric, no nans: ", accuracy)
```
Accuracy on sample data - numeric, no nans: 0.62

9)

```
# Import the Imputer object
from sklearn.preprocessing import Imputer

# Create training and test sets using only numeric data
X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric',
'with_missing']],
                                    pd.get_dummies(sample_df['label']),
                                    random_state=456)

# Insantiate Pipeline object: pl
pl = Pipeline([
```

```
        ('imp', Imputer()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

# Fit the pipeline to the training data
pl.fit(X_train,y_train)

# Compute and print accuracy
accuracy = pl.score(X_test,y_test)
print("\nAccuracy on sample data - all numeric, incl nans: ", accuracy)
```

```
accuracy on sample data - all numeric, incl nans:  0.636
```

Chapter 3)

1)
Preprocessing text features

Here, you'll perform a similar preprocessing pipeline step, only this time you'll use the `text` column from the sample data.
To preprocess the text, you'll turn to `CountVectorizer()` to generate a bag-of-words representation of the data, as in Chapter 2. Using the **default** arguments, add a `(step, transform)` tuple to the steps list in your pipeline.
Make sure you select only the text column for splitting your training and test sets.
As usual, your `sample_df` is ready and waiting in the workspace.

- Import `CountVectorizer` from `sklearn.feature_extraction.text`.
- Create training and test sets by selecting the correct subset of `sample_df`: `'text'`.
- Add the `'CountVectorizer'` step (with the name `'vec'`) to the correct position in the pipeline.

- Fit the pipeline to the training data and compute its accuracy.

```
# Import the CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
# Split out only the text data
X_train, X_test, y_train, y_test = train_test_split(sample_df['text'],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=456)

# Instantiate Pipeline object: pl
pl = Pipeline([
        ('vec', CountVectorizer()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
```

```
    ])

# Fit to the training data
pl.fit(X_train,y_train)

# Compute and print accuracy
accuracy = pl.score(X_test,y_test)
print("\nAccuracy on sample data - just text data: ", accuracy)
```

```
Accuracy on sample data - just text data:  0.808
```

2)

Multiple types of processing: FunctionTransformer

The next two exercises will introduce new topics you'll need to make your pipeline truly excel.

Any step in the pipeline *must* be an object that implements the `fit` and `transform` methods. The **`FunctionTransformer`** creates an object with these methods out of any Python function that you pass to it. We'll use it to help select subsets of data in a way that plays nicely with pipelines.

You are working with numeric data that needs imputation, and text data that needs to be converted into a bag-of-words. You'll create functions that separate the text from the numeric variables and see how the `.fit()` and `.transform()` methods work.

```
# Import FunctionTransformer
from sklearn.preprocessing import FunctionTransformer

# Obtain the text data: get_text_data
get_text_data = FunctionTransformer(lambda x: x['text'], validate=False)

# Obtain the numeric data: get_numeric_data
get_numeric_data = FunctionTransformer(lambda x: x[['numeric', 'with_missing']],
validate=False)

# Fit and transform the text data: just_text_data
just_text_data = get_text_data.fit_transform(sample_df)

# Fit and transform the numeric data: just_numeric_data
just_numeric_data = get_numeric_data.fit_transform(sample_df)

# Print head to check results
print('Text Data')
print(just_text_data.head())
print('\nNumeric Data')
print(just_numeric_data.head())
```

```
Text Data
0
1       foo
2    foo bar
3
4    foo bar
Name: text, dtype: object

Numeric Data
     numeric  with_missing
0 -10.856306      4.433240
1   9.973454      4.310229
2   2.829785      2.469828
3 -15.062947      2.852981
4  -5.786003      1.826475
```

3)

Multiple types of processing: FeatureUnion

Now that you can separate text and numeric data in your pipeline, you're ready to perform separate steps on each by nesting pipelines and using `FeatureUnion()`.

These tools will allow you to streamline all preprocessing steps for your model, even when multiple datatypes are involved. Here, for example, you don't want to impute our text data, and you don't want to create a bag-of-words with our numeric data. Instead, you want to deal with these separately and then join the results together using `FeatureUnion()`.

In the end, you'll still have only two high-level steps in your pipeline: preprocessing and model instantiation. The difference is that the first preprocessing step actually consists of a pipeline for numeric data and a pipeline for text data. The results of those pipelines are joined using `FeatureUnion()`

```
# Import FeatureUnion
from sklearn.pipeline import FeatureUnion

# Split using ALL data in sample_df
X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric',
'with_missing', 'text']],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=22)

# Create a FeatureUnion with nested pipeline: process_and_join_features
process_and_join_features = FeatureUnion(
```

```
        transformer_list = [
            ('numeric_features', Pipeline([
                ('selector', get_numeric_data),
                ('imputer', Imputer())
            ])),
            ('text_features', Pipeline([
                ('selector', get_text_data),
                ('vectorizer', CountVectorizer())
            ]))
        ]
    )

# Instantiate nested pipeline: pl
pl = Pipeline([
        ('union', process_and_join_features),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])


# Fit pl to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - all data: ", accuracy)
```

```
Accuracy on sample data - all data:  0.928
```

4)

Using FunctionTransformer on the main dataset

In this exercise you're going to use `FunctionTransformer` on the primary budget data, before instantiating a multiple-datatype pipeline in the next exercise. Recall from Chapter 2 that you used a custom function `combine_text_columns` to select and properly format **text data** for tokenization; it is loaded into the workspace and ready to be put to work in a function transformer!
Concerning the **numeric data**, you can use `NUMERIC_COLUMNS`, preloaded as usual, to help design a subset-selecting lambda function.
You're all finished with sample data. The original `df` is back in the workspace, ready to use.

```
# Import FunctionTransformer
from sklearn.preprocessing import FunctionTransformer

# Get the dummy encoding of the labels
```

```
dummy_labels = pd.get_dummies(df[LABELS])

# Get the columns that are features in the original df
NON_LABELS = [c for c in df.columns if c not in LABELS]

# Split into training and test sets
X_train, X_test, y_train, y_test =
multilabel_train_test_split(df[NON_LABELS],
                                                    dummy_labels,
                                                    0.2,
                                                    seed=123)

# Preprocess the text data: get_text_data
get_text_data = FunctionTransformer(combine_text_columns,validate=False)

# Preprocess the numeric data: get_numeric_data
get_numeric_data = FunctionTransformer(lambda x:
x[NUMERIC_COLUMNS], validate=False)
```

5)

Add a model to the pipeline

You're about to take everything you've learned so far and implement it in a `Pipeline` that works with the real, **DrivenData** budget line item data you've been exploring.
Surprise! The structure of the pipeline is exactly the same as earlier in this chapter:

- the **preprocessing step** uses `FeatureUnion` to join the results of nested pipelines that each rely on `FunctionTransformer` to select multiple datatypes
- the **model step** stores the model object

You can then call familiar methods like `.fit()` and `.score()` on the `Pipeline` object `pl`.

```
# Complete the pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
```

```
            ])),
            ('text_features', Pipeline([
                ('selector', get_text_data),
                ('vectorizer', CountVectorizer())
            ]))
        ]
    )),
    ('clf', OneVsRestClassifier(LogisticRegression()))
])

# Fit to the training data
pl.fit(X_train,y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

Accuracy on budget dataset: 0.203846153846

6)
Try a different class of model

Now you're cruising. One of the great strengths of pipelines is how easy they make the process of testing different models.
Until now, you've been using the model step `('clf', OneVsRestClassifier(LogisticRegression()))` in your pipeline.
But what if you want to try a different model? Do you need to build an entirely new pipeline? New nests? New FeatureUnions? Nope! You just have a simple one-line change, as you'll see in this exercise.
In particular, you'll swap out the logistic-regression model and replace it with a **random forest** classifier, which uses the statistics of an ensemble of decision trees to generate predictions

```
# Import random forest classifer
from sklearn.ensemble import RandomForestClassifier

# Edit model step in pipeline
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
```

```
                ('selector', get_text_data),
                ('vectorizer', CountVectorizer())
            ]))
        ]
    )),
    ('clf', RandomForestClassifier())
])

# Fit to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

Accuracy on budget dataset: 0.286538461538

7)

Can you adjust the model or parameters to improve accuracy?

You just saw a substantial improvement in accuracy by swapping out the model. Pipelines are amazing!
Can you make it better? Try changing the parameter `n_estimators` of `RandomForestClassifier()`, whose default value is `10`, to `15`.

```
# Import RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier

# Add model step to pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
            ]
```

```
        )),
        ('clf', RandomForestClassifier(n_estimators=15))
    ])

# Fit to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

Accuracy on budget dataset: 0.305769230769

Chapter 4)

1)

Deciding what's a word

Before you build up to the winning pipeline, it will be useful to look a little deeper into how the text features will be processed.
In this exercise, you will use `CountVectorizer` on the training data `X_train`(preloaded into the workspace) to see the effect of tokenization on punctuation.
Remember, since `CountVectorizer` expects a vector, you'll need to use the preloaded function, `combine_text_columns` before fitting to the training data.

```
# Import the CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

# Create the text vector
text_vector = combine_text_columns(X_train)

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate the CountVectorizer: text_features
text_features = CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Fit text_features to the text vector
text_features.fit(text_vector)

# Print the first 10 tokens
print(text_features.get_feature_names()[:10])
```

['00a', '12', '1st', '2nd', '3rd', '5th', '70', '70h', '8', 'a']

2)

N-gram range in scikit-learn

In this exercise you'll insert a `CountVectorizer` instance into your pipeline for the main dataset, and compute multiple n-gram features to be used in the model.
In order to look for ngram relationships at multiple scales, you will use the `ngram_range` parameter as Peter discussed in the video.
Special functions: You'll notice a couple of new steps provided in the pipeline in this and many of the remaining exercises. Specifically, the `dim_red` step following the `vectorizer` step , and the `scale` step preceeding the `clf`(classification) step. These have been added in order to account for the fact that you're using a reduced-size sample of the full dataset in this course. To make sure the models perform as the expert competition winner intended, we have to apply a **dimensionality reduction** technique, which is what the `dim_red` step does, and we have to **scale the features** to lie between -1 and 1, which is what the `scale`step does.
The `dim_red` step uses a scikit-learn function called `SelectKBest()`, applying something called the **chi-squared test** to select the K "best" features. The `scale` step uses a scikit-learn function called `MaxAbsScaler()` in order to squash the relevant features into the interval -1 to 1.

```
# Import pipeline
from sklearn.pipeline import Pipeline

# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

# Import CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

# Import other preprocessing modules
from sklearn.preprocessing import Imputer
from sklearn.feature_selection import chi2, SelectKBest

# Select 300 best features
chi_k = 300

# Import functional utilities
from sklearn.preprocessing import FunctionTransformer, MaxAbsScaler
from sklearn.pipeline import FeatureUnion

# Perform preprocessing
get_text_data = FunctionTransformer(combine_text_columns, validate=False)
```

```
get_numeric_data = FunctionTransformer(lambda x:
x[NUMERIC_COLUMNS], validate=False)

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer',
CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                            ngram_range=(1,2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
             ]
        )),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Note)

Interaction terms: the math

$$\beta_1 x_1 + \beta_2 x_2 + \beta_3 (x_1 \times x_2)$$

X1	X2	X3
0	1	X1*X2 = 0*1 = 0
1	1	X1*X2 = 1*1 = 1

Sparse interaction features

```
In [5]: SparseInteractions(degree=2).fit_transform(x).toarray()
Out[5]:
array([[ 0.,  1.,  0.],
       [ 1.,  1.,  1.]])
```

- The number of interaction terms grows exponentially
- Our vectorizer saves memory by using a sparse matrix
- PolynomialFeatures does not support sparse matrices
- We have provided SparseInteractions to work for this problem

3)

Implement interaction modeling in scikit-learn

It's time to add interaction features to your model.
The `PolynomialFeatures`object in scikit-learn does just that, but here you're going to a custom interaction object, `SparseInteractions`. Interaction terms are

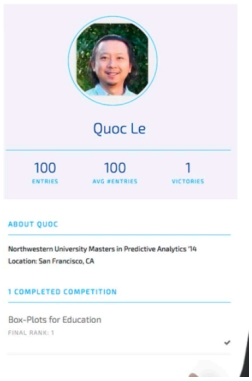
a statistical tool that lets your model express what happens if two features appear together in the same row.
`SparseInteractions` does the same thing as `PolynomialFeatures`, but it uses sparse matrices to do so. You can get the code for `SparseInteractions` at **this GitHub Gist**.
`PolynomialFeatures` and `SparseInteractions` both take the argument `degree`, which tells them what polynomia degree of interactions to compute.
You're going to consider interaction terms of `degree=2` in your pipeline. You will insert these steps *after* the preprocessing steps you've built out so far, but *before* the classifier steps.
Pipelines with interaction terms take a while to train (since you're making n features into n-squared features!), so as long as you set it up right, we'll do the heavy lifting and tell you what your score is!

```
# Instantiate pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer',
CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                            ngram_range=(1, 2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
            ]
        )),
        ('int', SparseInteractions(degree=2)),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Note)

The model that won it all

- You now know all the expert moves to make on this dataset
  - NLP: Range of n-grams, punctuation tokenization
  - Stats: Interaction terms
  - Computation: Hashing trick
- What class of model was used?



4)

Why is hashing a useful trick?

In the video, Peter explained that a **hash** function takes an input, in your case a token, and outputs a hash value. For example, the input may be a string and the hash value may be an integer.
We've loaded a familiar python datatype, a dictionary called `hash_dict`, that makes this mapping concept a bit more explicit. In fact, **python dictionaries ARE hash tables**!
Print `hash_dict` in the IPython Shell to get a sense of how strings can be mapped to integers.
By explicitly stating how many possible outputs the hashing function may have, we limit the size of the objects that need to be processed. With these limits known, computation can be made more efficient and we can get results faster, even on large datasets.
Using the above information, answer the following:
Why is hashing a useful trick?

Possible Answers

- Hashing isn't useful unless you're working with numbers. — press 1
- Some problems are memory-bound and not easily parallelizable, but hashing parallelizes them. — press 2
- Some problems are memory-bound and not easily parallelizable, and hashing enforces a fixed length computation instead of using a mutable datatype (like a dictionary). — press 3
- Hashing enforces a mutable length computation instead of using a fixed length datatype, like a dictionary. — press 4

5)
Implementing the hashing trick in scikit-learn

In this exercise you will check out the scikit-learn implementation of `HashingVectorizer` before adding it to your pipeline later.
As you saw in the video, `HashingVectorizer` acts just like `CountVectorizer`in that it can accept `token_pattern` and `ngram_range` parameters. The important difference is that it creates hash values from the text, so that we get all the computational advantages of hashing!

```
# Import HashingVectorizer
from sklearn.feature_extraction.text import HashingVectorizer

# Get text data: text_data
text_data = combine_text_columns(X_train)

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate the HashingVectorizer: hashing_vec
hashing_vec =
HashingVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Fit and transform the Hashing Vectorizer
hashed_text = hashing_vec.fit_transform(text_data)

# Create DataFrame and print the head
hashed_df = pd.DataFrame(hashed_text.data)
print(hashed_df.head())
```

```
          0
0 -0.160128
1  0.160128
2 -0.480384
3 -0.320256
4  0.160128
```

6)
Build the winning model

You have arrived! This is where all of your hard work pays off. It's time to build the model that won DrivenData's competition.
You've constructed a robust, powerful pipeline capable of processing training *and* testing data. Now that you understand the data and know all of the

tools you need, you can essentially solve the whole problem in a relatively small number of lines of code. Wow!
All you need to do is add the `HashingVectorizer` step to the pipeline to replace the `CountVectorizer` step.
The parameters `non_negative=True`, `norm=None`, and `binary=False` make the `HashingVectorizer` perform similarly to the default settings on the `CountVectorizer` so you can just replace one with the other.

```
# Import the hashing vectorizer
from sklearn.feature_extraction.text import HashingVectorizer

# Instantiate the winning model pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer',
HashingVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                              non_negative=True, norm=None,
binary=False,
                                              ngram_range=(1,2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
            ]
        )),
        ('int', SparseInteractions(degree=2)),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```