```
import lbpwv, os
bucket = lbpwv.Bucket('comp3321[redacted])
```

P.L. 86-36

```
filename = "nb_classification_data.csv"

if not filename in os.listdir():
    file_content = bucket.get(filename).content
    open(filename, 'wb').write(file_content)
features_labels = pd.read_csv(filename).set_index('id')

def preview(df, name='data', nrows=3, sampled=True):
    if sampled:
        df = df.sample(n=nrows)
    print("Preview of {}:".format(name))
    display(df.head(nrows))

preview(features_labels)
print("Statistical summary of data:")
display(features_labels.describe())
print("Datatypes in dataset:")
display(features_labels.dtypes)
```

Split Out Features and Labels

First we want to split out the features, which we'll use to predict the labels, from the actual labels. We'll also drop any features that will not be used to make the predictions. In this case we'll drop the 'notebook_url' column.

```
def split_features_labels(df):
    features_labels = df.copy()
    # change NaN notebook_text values to ''
    features_labels.loc[(features_labels.notebook_text != features_labels.notebook_text), 'notebook_text'] = ''
    # drop category and notebook_url for features
    features = features_labels.drop(['category', 'notebook_url'], axis=1)
    # split out the labels
    labels = features_labels.query("category == category").loc[:, ['category']]
    return features, labels

features, labels = split_features_labels(features_labels)

preview(features, name='features', sampled=False)
preview(labels, name='labels', sampled=False)
```

Normalize/Prepare Features

At their core, machine learning models are just mathematical algorithms, ranging from simple to very complex. One thing they all share in common is they work on numerical data. This presents a challenge when we want to use text-based or categorical features, such as the markdown text of a notebook or the name of the programming language in which the notebook was written. Fortunately there are multiple methods we can employ to convert this non-numerical data into numerical data.

Even with numerical data, we will often want to transform that data, by normalizing or scaling it (keeping all numerical data on the same scale) or engineering it in some way to make it more relevant. We'll show some examples of all of these below.

Prepare Numerical/Categorical Features

Create boolean feature indicating whether notebook is classified

This will convert our text classification data into numeric data: 0 for unclassified, 1 for confidential, 2 for secret, and 3 for top secret.

```
def classification_to_level(features):
    if 'classification' in features.columns:
        features['classification_level'] = features['classification'].apply(lambda x: x.split("/")[0])
        class_mappings = {'TOP SECRET': 3, 'SECRET': 2, 'CONFIDENTIAL': 1, 'UNCLASSIFIED': 0}
        features['classification_level'] = features['classification_level'].apply(lambda x: class_mappings[x])
        return features.drop(['classification'], axis=1)
    return features

features = classification_to_level(features)
preview(features)
```

Convert dates to number of days ago

Our model can't interpret timestamp data either, so we'll convert it into number of days ago. In this case, all of our data was uploaded on November 14, 2017, so we will calculate the number of days the notebooks were created and updated before that cut-off date.

```
from datetime import datetime, timedelta

def encode_datetime(x):
    strp_string = '%Y-%m-%d %H:%M:%S'
    if len(x) == 25:
        strp_string += ' +0000'
    timestamp = datetime.strptime(x, strp_string)
    return (datetime(2017, 11, 14) - timestamp).days

def datetime_to_days_ago(features, timezone=True):
    if 'created_at' in features.columns and 'updated_at' in features.columns:
        features['days_ago_created'] = features['created_at'].apply(lambda x: encode_datetime(x))
        features['days_ago_updated'] = features['updated_at'].apply(lambda x: encode_datetime(x))
        return features.drop(['created_at', 'updated_at'], axis=1)
    return features

features = datetime_to_days_ago(features)

preview(features)
```

Compute number of unique runs/downloads/views to total runs/downloads/views

This is another value judgment we're making--does the ratio of unique runs to total runs tell us something about the notebook? Could it be that a notebook with a lot of runs but few unique runs mean that it's more likely to be a mission notebook (i.e. mission users run the notebook over and over again)? This is where domain knowledge becomes key in machine learning. Understanding the data will help you engineer features that are more likely to be meaningful for the machine learning model.

```
def calc_ratios(features):
    features['unique_runs_to_runs'] = features['unique_runs'] / features['runs']
    features['unique_downloads_to_downloads'] = features['unique_downloads'] / features['downloads']
    features['unique_views_to_views'] = features['unique_views'] / features['views']
    features['updated_to_created'] = features['days_ago_updated'] / features['days_ago_created']
    return features
features = calc_ratios(features)
preview(features)
```

Scale features

Scaling features is very important in machine learning, and refers to putting all of your data on the same scale. Commonly this is a 0 to 1 scale, or it could also be in terms of standard deviations. This prevents large numeric values from being interpreted as more important than small numeric values. For example, you might have over 1,000 views on a particular notebook, but only 5 stars. Should the views feature be interpreted as being 200 times more important than the number of stars? Probably not. Scaling the features puts them all on a level playing field, so to speak.

```
from sklearn.preprocessing import MinMaxScaler

def scale_features(features, features_to_scale):
    scaler = MinMaxScaler()
    scaled_features = features.copy()
    scaler.fit(features[features_to_scale])
    scaled_features[features_to_scale] = scaler.transform(features[features_to_scale])
    return scaler, scaled_features

features_to_scale = [
  'views',
  'unique_views',
  'runs',
  'unique_runs',
  'downloads',
  'unique_downloads',
  'stars',
  'days_ago_created',
  'days_ago_updated',
  'classification_level'
]
scaler, scaled_features = scale_features(features, features_to_scale)
preview(scaled_features)
```

Impute missing values

Imputing means filling in missing values with something else. Commonly this can be a zero or the mean value for that column. We do the latter here.

```
from sklearn.impute import SimpleImputer

def impute_missing(scaled_features, features_to_impute):
    imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
    imputed_features = scaled_features.copy()
    imputer.fit(scaled_features[features_to_impute])
    imputed_features[features_to_impute] = imputer.transform(scaled_features[features_to_impute])
    imputed_features.loc[(imputed_features.notebook_text != imputed_features.notebook_text), 'notebook_text'] = ''
    return imputer, imputed_features

features_to_impute = [
  'health',
  'trendiness',
  'unique_runs_to_runs',
  'unique_downloads_to_downloads',
  'unique_views_to_views',
  'updated_to_created'
]

imputer, imputed_features = impute_missing(scaled_features, features_to_impute)

preview(imputed_features)
```

Encode categorical features

A common way to encode categorical features is to use what's called 'one-hot encoding'. This means taking a single column with multiple values and turning it into multiple columns with a boolean value (0 or 1) indicating the presence of that value. For example, for the column 'owner_type', we have two possibilities: 'Group' and 'User'. One-hot encoding turns this into two columns, 'owner_type_Group' and 'owner_type_User', where the possible values for each are 0 or 1. The 1 indicates the value is present for that row and the 0 indicates it is not.

```
def encode_categories(imputed_features, features_to_encode):
    encoded_features = imputed_features.copy()
    encoded_features = pd.get_dummies(imputed_features, columns=features_to_encode)
    return encoded_features

features_to_encode = ['lang', 'owner_type']
encoded_features = encode_categories(imputed_features, features_to_encode)
encoded_columns = [col for col in list(encoded_features) if re.sub("\_[A-Za-z]+$","",col) in features_to_encode]
preview(encoded_features)
```

Perform feature selection on non-text features

Sklearn has some algorithms built in to help identify the most important features to keep for the model. This can help the model perform better by removing some of the noise (features that don't have much predictive power) and also help it run more quickly. Below we're using the `SelectKBest` algorithm, which simply keeps the top `k` features according to the predictive power of that feature.

```
from sklearn.feature_selection import SelectKBest
from operator import itemgetter

import warnings
from sklearn.exceptions import DataConversionWarning
warnings.simplefilter('ignore', DataConversionWarning)

def select_kbest(encoded_features, k, text_features):
    skb = SelectKBest(k=k)
    numerical_features = encoded_features.drop(text_features, axis=1)
    skb.fit(numerical_features, labels)
    feature_scores = []
    selected_features = []
    for feature, score in zip(numerical_features.columns, skb.scores_):
        feature_scores.append({'feature': feature, 'score': score})
    feature_scores.sort(key=itemgetter('score'), reverse=True)

    for counter, score in enumerate(feature_scores):
        output_text = "Feature: {0}, Score: {1}".format(score['feature'], score['score'])
        if counter < k:
            output_text += " (selected)"
        selected_features.append(score['feature'])
        print(output_text)

    return selected_features

text_features = ['title', 'description', 'notebook_text']
joined_to_label = labels.join(encoded_features, how='left').drop('category', axis=1)
selected_features = select_kbest(joined_to_label, k=21, text_features=text_features)
```

Prepare Text Features

For preparing the text, we first will clean/normalize the text, then stem it, and then vectorize it with TF-IDF (Term Frequency-Inverse Document Frequency) weighting. These are all explained below.

Strip portion markings from description and notebook_text

Since we already have the classification level of the notebook as its own feature, we'll strip out the portion markings from the description and notebook_text fields.

```
import re

# strip portion markings from descriptions
def strip_portion(text):
    return re.sub('(?:\([\w/\s\,\-]*\))', '', text)

def strip_portion_markings(encoded_features):
    encoded_features['description'] = encoded_features['description'].apply(lambda x: strip_portion(x))
    encoded_features['notebook_text'] = encoded_features['notebook_text'].apply(lambda x: strip_portion(x))
    return encoded_features
stripped_pms = strip_portion_markings(encoded_features)
preview(stripped_pms)
```

Strip punctuation and numbers from text

Here's some more cleaning of the text data to strip out characters we don't care about. Specifically we're stripping out punctuation and numbers to keep just the words. There are scenarios where you might want to keep some of these things, but even then you would usually encode any number as [NUMBER] or something like that.

```
no_word_chars = re.compile('^[^a-z]+$')
strip_punct_digits = re.compile('[^\sa-z]')
strip_extra_spaces = re.compile('\s{2,}')
contains_digits = re.compile('(.*\d+.*)')
strip_punct = re.compile('[^\s\w]')

def normalize_text(text):
    normal = text.lower()
    # remove any words containing digits
    normal = " ".join([contains_digits.sub(' ', word) for word in normal.split(" ")])
    # remove words that contain no word characters(a-z)
    normal = " ".join([no_word_chars.sub(' ', word) for word in normal.split(" ")])
    # remove all punctuation and digits
    normal = strip_punct_digits.sub('', normal)
    # remove all punctuation
    #normal = strip_punct.sub(' ', normal)
    # replace consecutive spaces with a single space
    normal = strip_extra_spaces.sub(' ', normal)
    # remove leading and trailing whitespace
    normal = normal.strip()
    return normal

def normalize_text_features(df, text_features):
    temp_df = df.copy()
    for feature in text_features:
        temp_df[feature] = temp_df[feature].apply(lambda x: normalize_text(x))
    return temp_df

normalized_text = normalize_text_features(stripped_pms, ['description', 'title', 'notebook_text'])
preview(normalized_text)
```

Stem text

Stemming means stripping words down to their roots or stems, so that variations of similar words are lumped together. For example, we would judge 'played', 'play', 'plays', and 'player' to all be fairly simple, but if we vectorize them as separate words, our machine learning algorithm will treat them as completely separate. In order to keep that relationship (and reduce the number of features for the model to train on), we'll reduce these terms down to the common stem. A lot of machine learning on text features now uses word embeddings as a way to show relationships among terms, but that's outside the scope of this notebook.

```
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer('english')

def stem_text(df, text_features):
    temp_df = df.copy()
    for feature in text_features:
        temp_df[feature] = temp_df[feature].apply(lambda x: " ".join([stemmer.stem(word) for word in x.split(" ")]))
    return temp_df

stemmed = stem_text(normalized_text, ['description', 'title', 'notebook_text'])

preview(stemmed)
```

Transform Text Features, Combine with Numerical Features

We use a `DataFrameMapper` object from the `sklearn-pandas` library to map all of the text and numerical features together. The text features are first vectorized and weighted using TFIDF, which makes it so that terms seen commonly in a lot of the documents (notebooks in this case) are weighted lower to give them less importance. This prevents terms like `the`, `and`, etc., from being given too much importance.

```
# use DataFrameMapper to combine our text feature vectors with our numerical data
from sklearn_pandas import DataFrameMapper
from sklearn.feature_extraction.text import TfidfVectorizer

vect = TfidfVectorizer(stop_words='english', min_df=2)

mapper = DataFrameMapper([
    ('title', vect),
    ('description', vect),
    ('notebook_text', vect),
    (selected_features, None)
], sparse=True)

# split out labeled and unlabeled data; fit, train, test from the former, predict on the latter
joined_data = stemmed.join(labels, how='left')
training_data = joined_data.query("category == category").drop("category", axis=1)

# fit and transform the labeled data to build the model
mapper.fit(training_data[selected_features + ['title', 'description', 'notebook_text']])
training_features = mapper.transform(training_data[selected_features + ['title', 'description', 'notebook_text']])
```

Split Training and Testing Data

Now we split our data into a training and a test set, which is important to be able to test the success of our model. Here we keep 80% for training and 20% for testing.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(training_features, labels, test_size=0.2, random_state=123)

print("Training set has {} samples".format(X_train.shape[0]))
print("Testing set has {} samples".format(X_test.shape[0]))
```

Create Training and Predicting Pipeline

The below code just gives us a way to compare the performance of multiple models, in terms of training/run speed and accuracy /f1-score. The f1-score is a score that takes into account both precision (how many of my predictions of class A were actually class A) and recall (of those that were actually class A, how often did I predict those directly). This is especially important for imbalanced datasets. For example, if building a model to

predict fraudulent credit card transactions, a huge majority of credit card transactions are not likely to be fraudulent. If we built a model that just predicted that all transactions were not fraudulent, it might be correct over 99% of the time. But it would get a recall score of 0 on predicting transactions that were actually fraudulent. Using the f1-score would bring that overall score down accouting for that imbalance.

```
import matplotlib.patches as mpatches
from time import time
from sklearn.metrics import f1_score, accuracy_score, fbeta_score

def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
    """
    inputs:
        - learner: the learning algorithm to be trained and predicted on
        - sample_size: the size of samples (number) to be drawn from training set
        - X_train: features training set
        - y_train: labels training set
        - X_test: features testing set
        - y_test: labels testing set
    """

    results = {}

    # fit the learner to the training data
    start = time()
    learner.fit(X_train[:sample_size], y_train[:sample_size])
    end = time()

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

    # get predictions on test set
    start = time()
    predictions_test = learner.predict(X_test)
    end = time()

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

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

    # compute f-score on the test set
    results['f_test'] = f1_score(y_test, predictions_test, average='weighted')

    # Success
```

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

    # return the results
    return results

def evaluate(results):
    """
    Visualization code to display results of various learners.

    inputs:
        - learners: a list of supervised learners
        - stats: a list of dictionaries of the statistic results from 'train_predict()'

    """

    # create figure
    fig, ax = plt.subplots(1, 4, figsize=(14,4.5))

    # constants
    bar_width = 0.3
    colors = ['#A00000', '#00A0A0', '#00A000']

    # super loop to plot four panels of data
    for k, learner in enumerate(results.keys()):
        for j, metric in enumerate(['train_time', 'pred_time', 'acc_test', 'f_test']):
            for i in np.arange(2):
                # creative plot code
                ax[j].bar(i+k*bar_width, results[learner][i][metric], width=bar_width, color=colors[k])
                ax[j].set_xticks([0.45, 1.45])
                ax[j].set_xticklabels(["50%", "100%"])
                ax[j].set_xlabel("Training Set Size")
                ax[j].set_xlim((-0.1, 3.0))

    # add unique y-labels
    ax[0].set_ylabel("Time (in seconds)")
    ax[1].set_ylabel("Time (in seconds)")
    ax[2].set_ylabel("Accuracy Score")
    ax[3].set_ylabel("F-score")

    # add titles
    ax[0].set_title("Model Training")
```

```
    ax[1].set_title("Model Predicting")
    ax[2].set_title("Accuracy Score on Testing Set")
    ax[3].set_title("F-score on Testing Set")

    # set y-limits for score panels
    ax[2].set_ylim((0, 1))
    ax[3].set_ylim((0, 1))

    # create patches for the legend
    patches = []
    for i, learner in enumerate(results.keys()):
        patches.append(mpatches.Patch(color = colors[i], label = learner))
    plt.legend(handles = patches, bbox_to_anchor = (-1.55, -0.2), \
               loc = 'upper center', borderaxespad = 0., ncol = 3, fontsize = 'x-large')

    # aesthetics
    plt.suptitle("Performance Metrics for Three Supervised Learning Models", fontsize = 16, y = 1.10)
    plt.tight_layout()
    plt.show()
```

Evaluate Classification Models

Here we run the code to evaluate the different models.

```
# import three supervised learning models
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier
from sklearn.svm import SVC

# initialize the three models
clf_A = MultinomialNB()
clf_B = SVC(kernel='linear', random_state=123)
clf_C = RandomForestClassifier(random_state=123, n_estimators=100)

# calculate number of samples for 50% and 100% of the training data
samples_100 = len(y_train)
samples_50 = int(0.5 * samples_100)

# collect results on the learners
results = {}
for clf in [clf_A, clf_B, clf_C]:
    clf_name = clf.__class__.__name__
    results[clf_name] = {}
    for i, samples in enumerate([samples_50, samples_100]):
        results[clf_name][i] = train_predict(clf, samples, X_train, y_train, X_test, y_test)

# run metrics visualization for the three supervised learning models chosen
evaluate(results)
```

Model Tuning

Most models have a variety of what are called 'hyperparameters' that can be tuned to find the sets of hyperparameters that work better for predicting your data. Sklearn has a built-in object called GridSearchCV which lets you easily compare different sets of hyperparameters against each other and pick the settings that work the best, depending on the scoring function you choose. Be aware that adding lots of different hyperparameters can end up taking a really long time to calculate.

Doc ID: 6689695

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, fbeta_score, f1_score, accuracy_score

# initialize the classifier
#clf = SVC(kernel='linear', random_state=123, probability=True)
clf = RandomForestClassifier(random_state=123, n_estimators=100, bootstrap=True)

# create the list of parameters to tune
#parameters = {'C': [1.0, 5.0, 10.0], 'class_weight': [None, 'balanced']}
parameters = {
    'min_samples_split': [2, 3, 4]
}

# make an f1_score scoring object
scorer = make_scorer(f1_score, average='macro')

# perform grid search on the classifier
grid_obj = GridSearchCV(
    estimator=clf,
    param_grid=parameters,
    scoring=scorer
)

# fit the grid search to the training data
grid_fit = grid_obj.fit(X_train, y_train.values.ravel())

# get the best estimator
best_clf = grid_fit.best_estimator_

# make predictions using the unoptimized and best models
predictions = (clf.fit(X_train, y_train)).predict(X_test)
best_predictions = best_clf.predict(X_test)

# report the before and after scores
from sklearn.metrics import classification_report
print("Unoptimized model:")
print(classification_report(y_test, predictions))
print("Optimized model:")
print(classification_report(y_test, best_predictions))
```

```
# report the best parameters chosen
print("Best parameters for model: {}".format(best_clf.get_params()))
```

Extract Top Features

The Random Forest model which we used above gives you a relative importance score for each feature, which is indicating how useful the feature is in predicting the class. The scores are given as proportions, and all of them add up to 1.

```
name_to_importance = []

for name, importance in zip(mapper.transformed_names_, best_clf.feature_importances_):
    if name.startswith(selected_features[0]):
        index = int(name.split("_")[-1])
        name = selected_features[index]
    name_to_importance.append({'name': name, 'importance': importance})
name_to_importance.sort(key=itemgetter('importance'), reverse=True)
for counter, x in enumerate(name_to_importance[:25]):
    print("Rank: {}, Feature name: {}, Importance: {}".format(counter+1, x['name'], x['importance']))
```

Precision Scores Above Certain Probability

Here we'll just show how accurate the model (in terms of precision) above a certain probability. For example, for predictions made with a 0.9 or better probability score, how accurate was the model?

```
from collections import defaultdict

predicted_proba = best_clf.predict_proba(X_test)

max_probs = []
for index, value in enumerate(predicted_proba):
    probability = max(value)
    prediction = best_predictions[index]
    actual = y_test['category'].values[index]
    max_probs.append({'probability': probability, 'prediction': prediction, 'actual': actual})
max_probs_df = pd.DataFrame(max_probs)

# if we want to subset the data to above a certain probability, we can use the code below
#max_probs_df.query("probability > 0.8", inplace=True)

min_probability = max_probs_df['probability'].min()

max_probs_df['correct'] = max_probs_df['actual'] == max_probs_df['prediction']

possible_prob_scores = np.linspace(
  start=min_probability,
  stop=.95,
  num=int(np.ceil((.95 - min_probability) * 100))
)

prob_to_average_score = pd.DataFrame({'prob_score': possible_prob_scores})

average_scores = defaultdict(list)
for score in possible_prob_scores:
    average_score = np.mean(max_probs_df.query("probability >= {0}".format(
        round(score,3)
    ))['correct'])
    average_scores['total'].append(average_score)

prob_to_average_score['average_score'] = average_scores['total']

plt.figure(dpi=100, figsize=(10,5))
plt.plot(prob_to_average_score['prob_score'], prob_to_average_score['average_score'])
plt.xlabel("Probability cut-off (lower bound)")
plt.ylabel("Average accuracy percentage")
```

```
plt.title("Average accuracy percentage of predictions above a certain probability score")
plt.show()
```

Distribution of Probability Scores

This just shows the distribution of the probability scores. We want to see a skewed-left distribution, meaning that most of the predictions are made with a high probability score.

```
plt.figure(dpi=100, figsize=(10,5))
plt.hist(max_probs_df['probability'], bins=20)
plt.xlabel("Probability Scores")
plt.ylabel("Count")
plt.title("Distribution of Probability Scores from Predictions")
plt.show()
```

Show Stats on Categories of Notebooks

This is just showing the statistics on the categories of notebooks as of 14 November 2017--these are just based on the labels in our training data, not on the model predictions.

```
fig, ax = plt.subplots(figsize=(10,10))
labels['category'].value_counts().plot.pie(
  autopct='%1.1f%%',
  title="nbGallery Notebooks by Category as of 14 November 2017", ax=ax)
plt.show()
```

Run Model on a New Notebook

Just enter the URL for a notebook in nbGallery, and see what category the model predicts for it.

Doc ID: 6689695

```
class NotebookExtractor(object):

    def __init__(self):
        self.ses = requests_pki.Session()
        self.notebooks_url = 'https://nbgallery.nsa.ic.gov/notebooks/'
        self.tooltip_finder = re.compile("(?:notebook has been|health score)")

    def download_notebook(self, notebook_id):
        if "/" in notebook_id:
            notebook_id = notebook_id.rstrip("/")
            notebook_id = notebook_id.split("/")[-1]
        self.notebook_id = notebook_id.split("-")[0]
        resp = self.ses.get(
            self.notebooks_url + notebook_id,
            headers={
              'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20100101 Firefox/31.0',
              'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
            },
            timeout=10
        )
        return resp

    def to_soup(self, resp):
        return BeautifulSoup(resp.content)

    def extract_tooltips(self, soup):
        tooltips_dict = {}
        for link in soup.find_all('a', {'class': 'tooltips', 'title': self.tooltip_finder}):
            if " shared " in link['title']:
                continue
            if "health score" in link['title']:
                health = re.search("(\d+)\%", link['title']).groups()[0]
                tooltips_dict['health'] = round(float(health) / 100, 6)
            else:
                key = re.search("notebook has been (\w+) ", link['title']).groups()[0]
                key = re.sub("(r?ed)$", "", key)
                key += "s"
                total_val = re.search("(\d+)\stimes", link['title']).groups()[0]
                tooltips_dict[key] = int(total_val)
                unique_val = re.search("(\d+)\susers", link['title'])
```

```
                if unique_val:
                    tooltips_dict["unique_" + key] = int(unique_val.groups()[0])
        return tooltips_dict

    def extract_text(self, soup):
        extracted_text = []
        for a in soup.select("#notebookDisplay")[0].findAll():
            if a.name in ["h1", "h2", "h3", "h4", "h5", "p"]:
                extracted_text.append(a.text)
        return " ".join(extracted_text)

    def extract_description(self, soup):
        description = soup.find('meta', {'name': 'description'})['content']
        return description

    def extract_title(self, soup):
        title = soup.find('title').text
        return title

    def extract_classification(self, soup):
        classification = soup.find('div', {'class': re.compile("classBanner.+")}).text
        return classification

    def extract_owner_type(self, soup):
        group = soup.find('a', {'href': re.compile("\/groups\/.+")})
        if group:
            return 'Group'
        return 'User'

    def extract_lang(self, soup):
        lang_tag = soup.find('img', {'title': re.compile("This notebook is written in ")})
        lang = re.search("\s(\w+)$", lang_tag['title']).groups()[0].lower()
        return lang

    def extract(self, notebook_id, to_pandas=True):
        resp = self.download_notebook(notebook_id)
        soup = self.to_soup(resp)
        nb_dict = self.extract_tooltips(soup)
        nb_dict['notebook_text'] = self.extract_text(soup)
        nb_dict['description'] = self.extract_description(soup)
        nb_dict['title'] = self.extract_title(soup)
```

Doc ID: 6689695

```
        nb_dict['classification'] = self.extract_classification(soup)
        nb_dict['owner_type'] = self.extract_owner_type(soup)
        nb_dict['lang'] = self.extract_lang(soup)
        nb_dict['id'] = self.notebook_id
        if to_pandas:
            return pd.DataFrame([nb_dict]).set_index('id')
        return nb_dict

class NotebookModel(object):

    def __init__(self, df):
        self.df = df.copy()
        self.class_mappings = {'TOP SECRET': 3, 'SECRET': 2, 'CONFIDENTIAL': 1, 'UNCLASSIFIED': 0}
        self.scaler = scaler
        self.features_to_scale = features_to_scale
        self.imputer = imputer
        self.features_to_impute = features_to_impute
        self.features_to_encode = features_to_encode
        self.encoded_columns = encoded_columns
        self.validate_columns()
        self.initialize_regex()
        self.stemmer = SnowballStemmer('english')
        self.mapper = mapper
        self.selected_features = selected_features
        self.all_features = list(training_data)
        self.clf = best_clf

    def initialize_regex(self):
        self.pm_stripper = re.compile('(?:\([\w/\s\,\-]*\))')
        self.no_word_chars = re.compile('^[^a-z]+$')
        self.strip_punct_digits = re.compile('[^\sa-z]')
        self.strip_extra_spaces = re.compile('\s{2,}')
        self.contains_digits = re.compile('(.*\d+.*)')
        self.strip_punct = re.compile('[^\s\w]')

    def validate_columns(self):
        for col in self.features_to_scale + self.features_to_impute + self.features_to_encode:
            if col not in list(self.df):
                self.df[col] = np.nan

    def classification_to_level(self):
```

```
        self.df['classification_level'] = self.df['classification'].apply(lambda x: x.split("/")[0])
        self.df['classification_level'] = self.df['classification_level'].apply(lambda x: self.class_mappings[x])
        self.df.drop(['classification'], axis=1, inplace=True)

    def encode_datetime(self, x):
        strp_string = '%Y-%m-%d %H:%M:%S'
        if len(x) == 25:
            strp_string += ' +0000'
        timestamp = datetime.strptime(x, strp_string)
        return (datetime(2017, 11, 14) - timestamp).days

    def datetime_to_days_ago(self, timezone=True):
        if 'created_at' in self.df.columns and 'updated_at' in self.df.columns:
            self.df['days_ago_created'] = self.df['created_at'].apply(lambda x: self.encode_datetime(x))
            self.df['days_ago_updated'] = self.df['updated_at'].apply(lambda x: self.encode_datetime(x))
            self.df.drop(['created_at', 'updated_at'], axis=1, inplace=True)
        else:
            self.df['days_ago_created'] = 0
            self.df['days_ago_updated'] = 0

    def calc_ratio(self, field):
        if field in self.df.columns:
            self.df['unique_{0}_to_{0}'.format(field)] = self.df['unique_' + field] / self.df[field]
        else:
            self.df[field] = 0.0
            self.df['unique_' + field] = 0.0
            self.df['unique_{0}_to_{0}'.format(field)] = 0.0

    def calc_ratios(self):
        self.calc_ratio('runs')
        self.calc_ratio('downloads')
        self.calc_ratio('views')
        if self.df['days_ago_created'].values[0] == 0:
            self.df['updated_to_created'] = 0.0
        else:
            self.df['updated_to_created'] = self.df['days_ago_updated'] / self.df['days_ago_created']

    def scale_features(self):
        self.df[self.features_to_scale] = self.scaler.transform(self.df[self.features_to_scale])

    def impute_missing(self):
```

```
        self.df[self.features_to_impute] = self.imputer.transform(self.df[self.features_to_impute])

    def encode_categories(self):
        self.df = pd.get_dummies(self.df, columns=self.features_to_encode)
        for col in self.encoded_columns:
            if col not in list(self.df):
                self.df[col] = 0

    def strip_portion_markings(self):
        self.text_features = [col for col in list(self.df) if self.df[col].dtype == np.object]
        for col in self.text_features:
            self.df[col] = self.df[col].apply(lambda x: self.pm_stripper.sub('', x))

    def normalize_text(self, text):
        normal = text.lower()
        # remove any words containing digits
        normal = " ".join([self.contains_digits.sub(' ', word) for word in normal.split(" ")])
        # remove words that contain no word characters(a-z)
        normal = " ".join([self.no_word_chars.sub(' ', word) for word in normal.split(" ")])
        # remove all punctuation and digits
        normal = self.strip_punct_digits.sub('', normal)
        # remove all punctuation
        #normal = self.strip_punct.sub(' ', normal)
        # replace consecutive spaces with a single space
        normal = self.strip_extra_spaces.sub(' ', normal)
        # remove leading and trailing whitespace
        normal = normal.strip()
        return normal

    def normalize_text_features(self):
        for feature in self.text_features:
            self.df[feature] = self.df[feature].apply(self.normalize_text)

    def stem_text(self):
        for col in self.text_features:
            self.df[col] = self.df[col].apply(lambda x: " ".join([self.stemmer.stem(word) for word in x.split(" ")]))

    def fill_missing_columns(self):
        for feature in self.all_features:
            if feature not in list(self.df):
                self.df[feature] = 0
```

Doc ID: 6689695

```
            self.df = self.df.fillna(0)

    def transform(self):
        self.classification_to_level()
        self.datetime_to_days_ago()
        self.calc_ratios()
        self.scale_features()
        self.impute_missing()
        self.encode_categories()
        self.strip_portion_markings()
        self.normalize_text_features()
        self.stem_text()
        self.fill_missing_columns()
        self.transformed = self.mapper.transform(self.df)

    def predict(self):
        return {'predicted_class': self.clf.predict(self.transformed)[0]}

    def predict_proba(self):
        probs = self.clf.predict_proba(self.transformed)
        max_prob = round(np.max(probs), 3)
        max_class = self.clf.classes_[np.argmax(probs)]
        return {'predicted_class': max_class, 'probability': max_prob}


    def transform_predict(self, probability=False):
        self.transform()
        if probability:
            return self.predict_proba()
        return self.predict()

notebook_url = input("Enter the nbGallery URL for a notebook: ")

ne = NotebookExtractor()
nb_df = ne.extract(notebook_url)
model = NotebookModel(nb_df)
prediction = model.transform_predict(probability=True)
print("The predicted category for {} is {} with a probability of {}.".format(
    notebook_url, prediction['predicted_class'].title(), prediction['probability']))
```

COMP3321 Day02 Homework - GroceryDict.py

0

(b)(3)-P.L. 86-36

Created almost 3 years ago by [redacted] in COMP 3321

3 108 10

python exercises

(U) Homework for Day 2 of COMP3321.

Recommendations

(U) COMP3321 Day02 Homework - GroceryDict.py

Doc ID: 6689695

```
## Grocery List

myGroceryList = ["apples", "bananas", "milk", "eggs", "bread",
                 "hamburgers", "hotdogs", "ketchup", "grapes",
                 "tilapia", "sweet potatoes", "cereal",
                 "paper plates", "napkins", "cookies",
                 "ice cream", "cherries", "shampoo"]

vegetables = ["sweet potatoes", "carrots", "broccoli", "spinach",
              "onions", "mushrooms", "peppers"]
fruit = ["bananas", "apples", "grapes", "plumbs", "cherries", "pineapple"]
cold_items = ["eggs", "milk", "orange juice", "cheese", "ice cream"]
proteins = ["turkey", "tilapia", "hamburgers", "hotdogs", "pork chops", "ham", "meatballs"]
boxed_items = ["pasta", "cereal", "oatmeal", "cookies", "ketchup", "bread"]
paper_products = ["toilet paper", "paper plates", "napkins", "paper towels"]
toiletry_items = ["toothbrush", "toothpaste", "deodorant", "shampoo", "soap"]

GroceryStore = dict({"vegetables":vegetables, "fruit":fruit, "cold_items":cold_items,
                     "proteins":proteins, "boxed_items":boxed_items,
                     "paper_products":paper_products, "toiletry_items":toiletry_items})


myNewGroceryList = dict()
```

(U) Fill in your code below. Sort the items in myGroceryList by type into a dictionary: `myNewGrocerylist` . The keys of the dictionary should be:

```
["vegetables", "fruit", "cold_items",
 "proteins", "boxed_items", "paper_products",
 "toiletry_items"]
```

(U) Only use the `GroceryStore dict` , not the individual item lists, to do the sorting. Note: The keys for `myNewGroceryList` are the same as the keys for GroceryStore.

Doc ID: 6689695

```
print("My vegetable list: ", myNewGroceryList.setdefault('vegetables', list()))
print("My fruit list: ", myNewGroceryList.setdefault('fruit', list()))
print("My cold item list: ", myNewGroceryList.setdefault('cold_items', list()))
print("My protein list: ", myNewGroceryList.setdefault('proteins', list()))
print("My boxed item list: ", myNewGroceryList.setdefault('boxed_items', list()))
print("My paper product list: ", myNewGroceryList.setdefault('paper_products', list()))
print("My toiletry item list: ", myNewGroceryList.setdefault('toiletry_items', list()))
```

(U)

Password Project Instructions for COMP3321

0

(b)(3)-P.L. 86-36

Updated 6 months ago by [redacted] in COMP 3321

3 130 18

comp3321 password projects

(U) Password Project Instructions for COMP3321 [redacted]

Recommendations

P.L. 86-36

(U) Password Project Instructions for COMP3321

(U) These are the password project instructions for COMP3321 [redacted] You need to send a file containing your function(s) to the instructors, not a notebook. Files should be named SID_password_functions.py

(U) Password Checker Function

(U) Demonstrates the ability to loop over data, utilizing counters and checks to see if all requirements are met.

(U) Write a function called `password_checker` that takes as input a string, which is your password, and returns a boolean `True` if the password meets the following requirements and `False` otherwise.

1. Password must be at least 14 characters in length.
2. Password must contain at least one character from each of the four character sets defined below, and no other characters.
3. Passwords cannot contain more than three consecutive characters from the same character set as defined below.

(U) Character sets:

- Uppercase characters (`string.ascii_uppercase`)
- Lowercase characters (`string.ascii_lowercase`)
- Numerical digits (`string.digits`)
- Special characters (`string.punctuation`)

You may want to write multiple functions that your password checker function calls.

**Due: End of Day 3 **

```
def password_checker(password):
    """
    This is my awesome docstring for my awesome password
    checker that the author should adjust to say something
    more meaningful.
    """
    # Put code here
    return True
```

(U) Run the following bit of code to check your `password_checker` function. If your code is good, you should get four (4) `True` statements printed to the screen.

```
# This is a good password
print(password_checker("abcABC123!@#abcABC123!@#") == True)

# This is invalid because the runs of same character set are too long
print(password_checker("abcdefgABCDEFG1234567!@#$%^&") == False)

# This is invalid because there are no characters from string.punctuation
print(password_checker("abcABC123abcABC123") == False)

# This is invalid because it is too short
print(password_checker("aaBB11@@") == False)
```

(U) Password Generator Function

(U) Demonstrates the ability to randomly insert characters into a string that meets specific password requirements

(U) Write a function called `password_generator` that takes as **optional** argument the password length and returns a valid password, as defined in Password Checker Function. If no length is passed to the function, it defaults to 14. The following code outline **does not** account for the optional argument. You must make a change to that.

(U) Do not use the `password_checker` function in your `password_generator`. You can use it after you get something returned from your `password_generator` for your own testing, but it should not be part of the function itself.

Due: End of Day 5

```
def password_generator(length):
    """
    This is my awesome docstring for my awesome password
    generator that the author should adjust to say something
    more meaningful.
    """
    # Put code here
    return True
```

(U) Assuming you have a valid `password_checker` function, use the following code to check your `password_generator`. If no `False`s print, you are good. Otherwise, something is up.

```
my_password = password_generator()
if len(my_password) != 14 or not password_checker(my_password):
    print(False)

my_password = password_generator(25)
if len(my_password) != 25 or not password_checker(my_password):
    print(False)
```

(U) If you really want to test it out, run the following. If `False` prints, something is wrong.

```
from random import randint
for i in range(10000):
    if not password_checker(password_generator(randint(14,30))):
        print(False)
```

Final Project Schedule Generator

Updated 3 months ago by [redacted] in COMP 3321

3 10 4 1

fcx9 random python comp3321

(U) Little notebook for randomly generating a final project presentation schedule. Students will present every 30 minutes starting at the start time specified, with an optional hour blocked off for lunch.

Recommendations

Import Dependencies

```
import ipydeps
ipydeps.pip(['query_input'])

import random
from datetime import datetime, timedelta
import ipywidgets as widgets
import re
from IPython.display import display, clear_output
from query_input import QueryInput
```

Run Random Generator!

This uses a `RandomGenerator` class that inherits from the `QueryInput` class from the `query_input` package we imported to make creating the widget box and extracting the values out a little easier.

Doc ID: 6689697

```
class RandomGenerator(QueryInput):

    def __init__(self, title="Enter data for random project presentation time generator"):
        super(RandomGenerator, self).__init__(title)
        self.default_layout = {'l_width': '200px', 'r_width': '400px', 'r_justify_content': 'flex-end',
                               'box_justify_content': 'center'}

    def generate_times(self):
        start_times = []
        for i in range(900, 1500, 50):
            start_time = str(i)
            start_time = re.sub("50$","30",start_time)
            if len(start_time) == 3:
                start_time = '0' + start_time
            start_times.append(start_time)
        return start_times

    def random_schedule(self, students, start_time, lunch_time=None):
        start_time = datetime.strptime(start_time, "%H%M")
        if lunch_time:
            lunch_time = datetime.strptime(lunch_time, "%H%M")
        random.shuffle(students)
        for student in students:
            print(f"{student} will present at {start_time.strftime('%H%M')}")
            if lunch_time and start_time == lunch_time - timedelta(minutes=30):
                start_time += timedelta(minutes=90)
            else:
                start_time += timedelta(minutes=30)

    def submit(self, b):
        clear_output(wait=True)
        self.validate_input()
        self.extract_input()
        students = [student.strip() for student in self.extracted_input['student_names'].split("\n") if student.strip()]
        self.random_schedule(students, self.extracted_input['start_time'], self.extracted_input.get('lunch_time'))

    def create_input_form(self, start_times):
        self.build_box(description="<b/>Enter student names, one per line:",
                       name='student_names', required=True, widget_type=widgets.Textarea, r_height='400px',
                         **self.default_layout)
```

```
        self.build_box(description="<b/>Select the start time:", options=start_times, name='start_time',
                       required=True, widget_type=widgets.Select, **self.default_layout)
        self.build_box(description="<b/>Select a lunch time (optional):", options=[None] + start_times, name='lunch_time',
                       required=False, widget_type=widgets.Select, **self.default_layout)
        self.build_box(description='', name='submit', widget_type=widgets.Button, button_text='Submit',
                 button_click=self.submit, required=False, box_justify_content='center', r_border='solid 1px',
                r_width='200px', r_height='50px')

    def run(self):
        start_times = self.generate_times()
        self.create_input_form(start_times)
        self.display(border='solid 2px')

rg = RandomGenerator()
rg.run()
```