livelessons
video instruction from technology experts

Lesson 3: Your First Spark Application

3.11 Testing k-means with DonorsChoose.org Essays

```
def Kmeans(features, centers, num_iter):

    # track convergence
    error = []

    # compute closest centroid for given data point
    def closest_centroid(point, centroids):
        distances = [ np.sqrt(np.sum((point - c) ** 2)) for c in centroids ]
        return (np.argmin(distances), np.min(distances))

    # format return value appropriately for RDD
    def compute_assignments(point, centroids):
            closest = closest_centroid(point, centroids)
            return (closest[0], (point, 1, closest[1]))

    # iterate until convergence or total num_iter
    for i in range(num_iter):

        # assign each point to closest centroid
        assignments = features.map(lambda x: compute_assignments(x, centers))

        # update centroids to mean of assigned points
        means = assignments.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2]))
        centroids = means.map(lambda cent: (cent[0], cent[1][0]/cent[1][1])).collect()

        # map each new update to the appropriate position in centroid array
        for i, p in centroids:
            centers[i] = p

        # compute within-cluster error
        WSSE = means.map(lambda x: x[1][2]).sum()
        error.append(WSSE)

        # reached convergence?
        if len(error) > 1 and error[-2] == error[-1]:
            break

    return (assignments, centroids, error)
```

```
import nltk, string

def tokenize(text):
    tokens = []

    for word in nltk.word_tokenize(text):
        if word \
            not in nltk.corpus.stopwords.words('english') \
            and word not in string.punctuation \
            and word != '``':
                tokens.append(word)

    return tokens
```

```
# parse input essay file
essay_rdd = sc.textFile('file:///Users/jonathandinu/spark-ds-applications/data/donors_choose/essays.json')
row_rdd = essay_rdd.map(lambda x: json.loads(x))

# tokenize documents
tokenized_rdd = row_rdd.filter(lambda row: row['essay'] and row['essay'] != '') \
                       .map(lambda row: row['essay']) \
                       .map(lambda text: text.replace('\\n', '').replace('\r', '')) \
                       .map(lambda text: tokenize(text))

# compute term and document frequencies
term_frequency = tokenized_rdd.map(lambda terms: Counter(terms))
doc_freq_art = term_frequency.flatMap(lambda counts: counts.keys()) \
                             .map(lambda keys: (keys, 1)) \
                             .reduceByKey(lambda a, b: a + b)
```

```
num_features = 10000

# get most common terms to cut down on noise
top_terms = doc_freq_art.top(num_features, key=lambda a: a[1])
total_docs = tokenized_rdd.count()

# compute idf score
idf = map(lambda tup: (tup[0], math.log(float(total_docs) / tup[1])), top_terms)
broadcast_idf = sc.broadcast(idf)

# create tf-idf transformation
def vectorize(tokens):
    word_counts = Counter(tokens)
    doc_length = sum(word_counts.values())

    vector = [word_counts.get(word[0], 0) * word[1] / float(doc_length) for word in broadcast_idf.value]
    return np.array(vector)

# apply tf-idf weighting to tokens
bag_of_words = tokenized_rdd.filter(lambda x: len(x) > 0).map(vectorize)
bag_of_words.persist()
```

```
# initialize centroids
k = 100

centroids = bag_of_words.takeSample(False, k)
```

```
text_results = Kmeans(bag_of_words, centroids, 5)
```