Lecture-4-Feature-Extraction

May 7, 2019

1 Machine Learning

Lecture 4
Feature Extraction, Scikit Learn API, Model Evaluation

2 Overview

- Feature Extraction
- Scikit Learn API: Estimators, Pipelines
- Model Evaluation

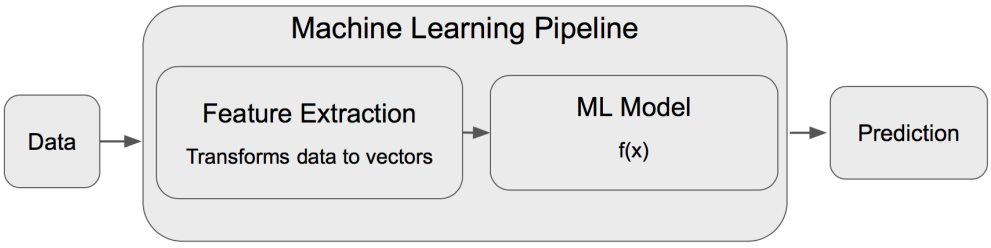
3 Machine Learning Pipelines

4 Scikit Learn API

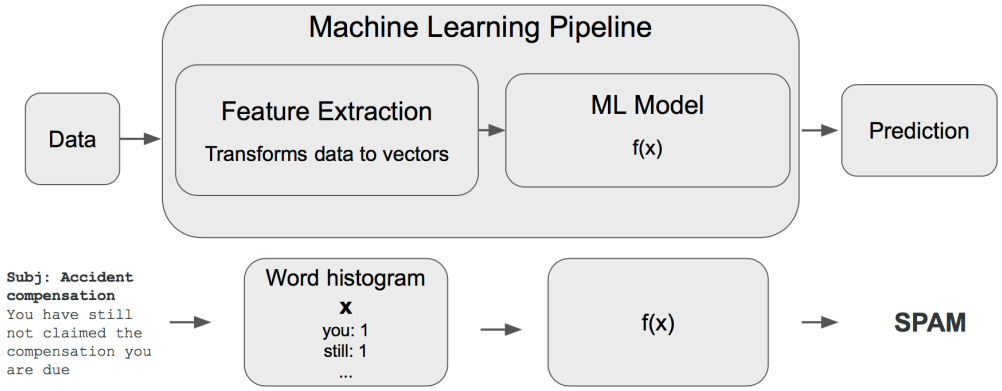
Scikit Learn is a popular machine learning library.
Its API was copied in many other libraries (e.g. spark.ml).
It offers most feature transformations, ML models, metrics and many useful tools.

4.1 Estimators

- implement `set_params()` and `get_params()`
- implement `fit` method
- Examples for what `fit` does:



ml-pipeline-2.png



ml-pipeline-2.png

  - sklearn.feature_extraction.text.CountVectorizer: learns dictionary
  - sklearn.linear_model.Perceptron: learns weight vector and Pipelines

- implement `transform` method
- Examples for what `transform` does:
  - sklearn.feature_extraction.text.CountVectorizer: extracts n-gram counts
  - sklearn.linear_model.Perceptron: projects data onto weight vector

5 Machine Learning Pipelines

5.1 Pipelines

Estimators with standardized API can be chained in a Pipeline.
This allows to chain all feature extraction and classification into one object
Pipelines behave like Estimators themselves, with `fit` and `transform` methods
All parameters of a Pipeline can be optimized jointly

```
In [1]: from sklearn.pipeline import Pipeline
        from sklearn.feature_extraction.text import CountVectorizer
        from sklearn.neighbors import NearestCentroid

        reviews = [
                "This is a horrible movie",
                "a great movie",
                "a fantastic movie",
            ]
        sentiment = [-1, 1, 1]

        text_clf = Pipeline([('vect', CountVectorizer()),
                             ('clf', NearestCentroid())])

        text_clf.fit(reviews, sentiment)

        text_clf.predict(reviews)
```

```
Out[1]: array([-1,  1,  1])

In [2]: text_clf.steps

Out[2]: [('vect', CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                  dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                  lowercase=True, max_df=1.0, max_features=None, min_df=1,
                  ngram_range=(1, 1), preprocessor=None, stop_words=None,
                  strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                  tokenizer=None, vocabulary=None)),
         ('clf', NearestCentroid(metric='euclidean', shrink_threshold=None))]

In [6]: text_clf.steps[0][1].transform(reviews)

Out[6]: <3x6 sparse matrix of type '<class 'numpy.int64'>'
                with 8 stored elements in Compressed Sparse Row format>

In [8]: text_clf.steps[0][1].transform(reviews).toarray()

Out[8]: array([[0, 0, 1, 1, 1, 1],
               [0, 1, 0, 0, 1, 0],
               [1, 0, 0, 0, 1, 0]])

In [10]: x = text_clf.steps[0][1].transform(reviews)
         text_clf.steps[1][1].predict(x)

Out[10]: array([-1,  1,  1])
```

6 Feature Extraction

- Feature extraction describes the transformations from **data** to **vectors**
- Extracting good features is **more important** than choosing a ML model
- We will cover some standard feature extractors, but there are many more
- Often the best feature extractors include domain knowledge by human experts
- Feature extractors have to be optimized along with all other model parameters

7 Feature Extraction

- Continuous Features
- Categorical Features
- Text
- Images

7.1 Continuous Features

- Continuous features: $x \in R^d$
- For many models continuous features don't need to be transformed
- For some models it is necessary or beneficial to **normalize** continuous features
  - When optimizing with stochastic gradient descent
  - When regularizing models: to control regularization

7.2 Normalization: z-scoring

Given a feature $x \in R^1$ (and for multivariate x analogously) there are several standard normalization options:

Standard scaling / z-scoring: compute mean μ_x and standard deviation σ_x of x and compute

$$x \leftarrow \frac{(x - \mu_x)}{\sigma_x}$$

- Resulting variable has zero mean and unit variance
- See sklearn.preprocessing.StandardScaler

```
In [1]: from sklearn.preprocessing import StandardScaler
        data = [[0, 0], [0, 0], [1, 1], [1, 1]]
        scaler = StandardScaler()
        scaler.fit(data)
        print("Estimated Mean: {}".format(scaler.mean_))
        scaler.transform([[.5, .5], [2,2]])
```

```
Estimated Mean: [0.5 0.5]
```

```
Out[1]: array([[0., 0.],
               [3., 3.]])
```

7.3 Normalization: Min-Max Scaling

Min-max scaling: compute min $\min_x$ and max $\max_x$ of x

$$x \leftarrow \frac{(x - \min_x)}{\max_x - \min_x}$$

- Resulting variable is in range [0,1] (or any other range)
- See sklearn.preprocessing.MinMaxScaler

7.4 Categorical Features

Categorical features are variables $x \in \{0, 1, 2, \ldots, N\}$ taking one value of a set of cardinality N without an implicit ordering e.g.: - colors (red, green, blue) - user ids - Movie ids

Often used feature transformations are

- One-hot encoding
- More recently for neural networks: low dimensional embeddings

7.5 One-hot encoding

Given a set of values [red, green, blue], we transform the data into

- red $\leftarrow [1,0,0]$
- green $\leftarrow [0,1,0]$
- blue $\leftarrow [0,0,1]$

Sets of categorical variables can be represented as sum over single value vectors.
Examples: bag-of-words vectors

```
In [2]: from sklearn.preprocessing import OneHotEncoder
        enc = OneHotEncoder()
        X = [['red'], ['green'], ['blue'], ['red']]
        enc.fit(X)
        enc.transform([['red'], ['green'], ['blue']]).toarray()

Out[2]: array([[0., 0., 1.],
               [0., 1., 0.],
               [1., 0., 0.]])
```

7.6 One-hot encoding: Problems

- Cardinality needs to be estimated upfront
- New items / categories cannot be represented
- Similarity information is lost (light-blue and blue as different as black and white)

8 Feature Extraction for Text

- Bag-of-Words
- Hashed Bag-of-Words
- (Continuous Bag-of-words)

8.1 Bag-of-Words Features

Count word (=token) occurrences
"if you pay peanuts , you get monkeys ."

get	if	monkeys	pay	peanuts	you	...
1	1	1	1	1	2	...

- Only accounts for word histogram
- Most of language structure lost
- Very efficient

8.2 Bag-of-Words Model with N-Grams

Count n-gram occurrences can account for some language structure

"if you pay peanuts , you get monkeys ."

get monkeys	if you	you pay	pay peanuts	...
1	1	1	1	...

```
In [14]: from sklearn.feature_extraction.text import CountVectorizer
         corpus = [
              'This is the first document.',
              'This document is the second document.',
              'And this is the third one.',
              'Is this the first document?']
         vectorizer = CountVectorizer()
         X = vectorizer.fit_transform(corpus)
         print("Vocabulary: {}".format(vectorizer.get_feature_names()))
         X.toarray()
```

```
Vocabulary: ['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

```
Out[14]: array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
                [0, 2, 0, 1, 0, 1, 1, 0, 1],
                [1, 0, 0, 1, 1, 0, 1, 1, 1],
                [0, 1, 1, 1, 0, 0, 1, 0, 1]], dtype=int64)
```

8.3 Character N-Grams

Instead of words as tokens you can take characters as tokens

"if you pay peanuts , you get monkeys ."

g	e	t	ge	et	...
1	1	1	1	1	...

Very powerful for language-independent models!

8.4 Term Frequency / Inverse Document Frequency

Frequently occurring words (like `the`) are often not very meaningful and are often weighted down by the **inverse document frequency**:

- Term frequency:

  $ tf(t,d) $: Frequency of a term t in document d

- Inverse document frequency:

  $\text{idf}(t,d) = \log \frac{|\text{Number of documents}|}{|\text{Number of Documents containing } t|}$

8.5 Alternatively: Stopwords

Just exclude some frequent meaningless words from the text.

8.6 N-Grams are expensive

V = number of tokens in your vocabulary (usually 1e4 to 1e6 for good models) - unigrams: all tokens

```
Memory requirement for n-gram counts: $V$
```

- bigrams: all ordered pairs of **two** tokens
  Memory requirement for n-gram counts: V^2
- trigrams: all ordered pairs of **three** tokens
  Memory requirement for n-gram counts: V^3
- ...

See Google's n-gram corpus (up to 5-grams for many languages).

8.7 Hashed N-Grams

- Instead of counting all n-grams: Use a HashMap
- Each token gets hashed, count of hash-bucket incremented
- Advantages:
  - Full control over memory footprint (often small HashMaps work surprisingly well)
  - No need to compute a dictionary
- Disadvantages:
  - Hash collisions: multiple n-gram map to the same bucket
  - No Tf-Idf weighting:

See Scikit-Learn's HashingVectorizer

8.8 Feature Extraction for Images

- Classical features (e.g. HOG)
- CNN features

8.9 Classical Computer Vision Feature Extractors

Histograms of Oriented Edges (HOG) descriptors capture shape very well.

HOG features, just like most other oldschool vision feature descriptors, perform poorly compared to neural networks

Still, they could be helpful if shape on standardized images is the only relevant feature

```
In [62]: from skimage.feature import hog
         from skimage import data, exposure, io

         # image = data.astronaut()
         image = io.imread("figures/turing.png")

         fd, hog_image = hog(image, orientations=8, pixels_per_cell=(16, 16),
                             cells_per_block=(1, 1), visualize=True, multichannel=True)
```

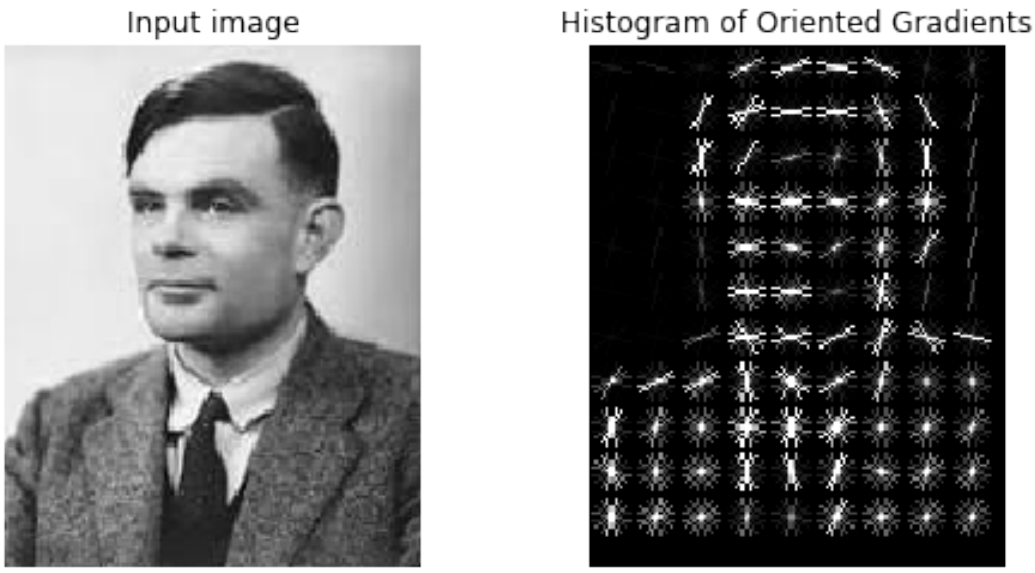
```
In [63]: import matplotlib.pyplot as plt
         %matplotlib inline

         fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), sharex=True, sharey=True)

         ax1.axis('off')
         ax1.imshow(image, cmap=plt.cm.gray)
         ax1.set_title('Input image')

         # Rescale histogram for better display
         hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

         ax2.axis('off')
         ax2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
         ax2.set_title('Histogram of Oriented Gradients')
         plt.show()
```



8.10 Convolutional Neural Networks

State of the art in computer vision are convolutional neural networks

We don't have time in this lecture to go into detail (see e.g. Kristian Hildebrand's lecture)

And most people don't have the compute to train those models from scratch

But pretrained models are good feature extractors - which doesn't require training

```
In [21]: from keras.applications.vgg19 import VGG19
         from keras.preprocessing import image
         from keras.applications.vgg19 import preprocess_input
         from keras.models import Model
         import numpy as np

         base_model = VGG19(weights='imagenet')
         model = Model(inputs=base_model.input, outputs=base_model.get_layer('block4_pool').ou

         img_path = 'figures/turing.png'
         img = image.load_img(img_path, target_size=(224, 224))
         x = image.img_to_array(img)
         x = np.expand_dims(x, axis=0)
         x = preprocess_input(x)

         block4_pool_features = model.predict(x)
         print("image dimensions: {}".format(x.shape))
         print("image feature dimensions: {}".format(block4_pool_features.shape))
```

```
image dimensions: (1, 224, 224, 3)
image feature dimensions: (1, 14, 14, 512)
```

```
In [86]: def plot_channel(img, channel, ax):
             ax.axis('off')
             ax.imshow(img, cmap=plt.cm.gray)
             ax.set_title('Channel {}'.format(channel))

         channels = [3, 56, 170]

         _, axs = plt.subplots(1, len(channels), figsize=(12, 4), sharex=True, sharey=True)

         for channel, ax in zip(channels, axs):
             plot_channel(block4_pool_features[0,:,:,channel], channel, ax)
```