特征工程

参考美团三篇博客,据说很牛https://tech.meituan.com/waimai_data_feature_service.html
https://tech.meituan.com/feature_pipeline.html
https://tech.meituan.com/machinelearning-data-feature-process.html
以及http://www.cnblogs.com/jasonfreak/p/5448385.html

1.特征：从数据中抽取出来的对结果预测有用的信息

特征工程是使用专业背景知识和技巧处理数据，是的特征在机器学习算法上发挥更好的作用的过程
更好的特征以为着更强的灵活性，以为着只需要使用简单的模型，以为着更好的结果

2.数据采集，格式化，数据清洗

清洗：包括脏数据的删除，缺省值得填充等等

3.数据采样：

很多情况下正负样本是不均衡的，但大多数模型对正负样本比是敏感的，这就需要进行采样，如随机采样，分层采样

正负样本不均衡的处理办法：、

-----正样本>>负样本，且数据量都很大 ==》下采样(downsampling),按照负样本的数量从正样本里采样到与负样本1:1

-----正样本>>负样本，但量不大 ==》1）采集更多数据 2）上采样(oversampling) ，对负样本进行重复采样 3)修改损失函数

4.特征处理

包括数值型，类别型，时间型，文本型，统计型，组合特征等等

a.数值型

去量纲

标准化:对每一维特征计算全体样本的均值和方差,然后将特征值缩放为:

$$x' = \frac{x - \overline{X}}{S}$$

标准化不改变原始的概率分布,只是进行缩放,去量纲
使用preproccessing库的StandardScaler类对数据进行标准化的代码如下：

```
from sklearn.preprocessing import StandardScaler
StandardScaler().fit_transform(iris.data)
```

归一化:规则为l2的归一化公式如下:

$$x' = \frac{x}{\sqrt{\sum_j^m x[j]^2}}$$

归一化的操作对象是行,考虑的是一个样本的所有特征,其目的在于样本向量在点乘运算或他核函数计算相似性时，拥有统一的标准，也就是说都转化为“单位向量”。
使用preproccessing库的Normalizer类对数据进行归一化的代码如下：

```
from sklearn.preprocessing import Normalizer
Normalizer().fit_transform(iris.data)
```

区间缩放:区间缩放法的思路有多种，常见的一种为利用两个最值进行缩放，公式表达为：

$$x' = \frac{x - Min}{Max - Min}$$

```
from sklearn.preprocessing import MinMaxScaler
MinMaxScaler().fit_transform(iris.data)
```

统计值max，min，mean，std

离散化:

给连续值赋予非线性特性。划分为n个区间，某样本在某区间出现，则向量对应位置值为1，其余位置为0

```
data = numpy.random.randint(0,10,100) //生成100个随机数
data = pd.cut(data,4) //把数据划分成四段,输出每个数据属于哪个区间,
                      //还有一个qcut方法,按分位数切分区间
data = pd.get_dummies(data) //用One-Hot编码为四维向量
```

还有一种方式比较粗暴,就是二值化:

```
from sklearn.preprocessing import Binarizer
Binarizer(threshold=3).fit_transform(iris.data)
```

b.类别型

one-hot编码--->哑变量

```
pd.get_dummies(data)
```

柱状分布:

统计每个类别下的样本数量,按其在总体样本中所占的比例进行编码

```
data = numpy.random.randint(0,10.100)
data = pd.value_counts(data) //统计各取值供有多少个样本,输出其样本数
data = data/100 //算出取该值的样本数量占总样本数的比例,以这个比例作为该取值的编码
```

Hash trick:

当特征维度很大时,需要进行降维处理,通过Hash函数把n维特征转为m维.具体的方法是，对应任意一个特征名，我们会用Hash函数找到对应哈希表的位置，然后将该特征名对应的词频统计值累加到该哈希表位置。如果用数学语言表示,假如哈希函数h使第i个特征哈希到位置j,即h(i)=j,则第i个原始特征的词频数值 (i)将累加到哈希后的第j个特征的词频数值 (j)即：

$$\bar{\phi}(j) = \sum_{i \in \mathcal{J}; h(i)=j} \phi(i)$$

其中$\mathcal{J}$是原始特征的维度。

但是上面的方法有一个问题，有可能两个原始特征的哈希后位置在一起导致词频累加特征值突然变大，为了解决这个问题，出现了hash Trick的变种**signed hash trick**,此时除了哈希函数 h,我们多了一个一个哈希函数：

$$\xi : \mathbb{N} \to \pm 1$$

此时我们有:

$$\bar{\phi}(j) = \sum_{i \in \mathcal{J}; h(i)=j} \xi(i)\phi(i)$$

哈希后的特征仍然是一个无偏的估计，不会导致某些哈希位置的值过大.

在scikit-learn的HashingVectorizer类中，实现了基于signed hash trick的算法

```
from sklearn.feature_extraction.text import HashingVectorizer
vectorizer2=HashingVectorizer(n_features = 6,norm = None)
print vectorizer2.fit_transform(corpus)
```

Histogram映射

根据某一个特征或者预测值来反编码当前特征（统计每个类别变量下各个target比例，转成数值）

男	21	…	足球
男	48	…	散步
女	22	…	看电视剧
男	21	…	足球
女	30	…	看电视剧
女	50	…	散步

男:[2/3,1/3,0]; 女:[0,1/3,2/3]:男生中有2/3爱踢足球,1/3爱散步,0爱看电视剧,女生中有0爱踢足球,1/2爱散步,2/3爱看电视剧

同理,可以通过target值反编码年龄:

21:[1,0,0];22:[0,0,1]

c.时间型

时间型特征比较特殊,既可以看做连续值,也可以看做离散值

连续值:

持续时间(某网页单页浏览时间)

间隔时间(上次购买/点击离现在的时间)

离散值:

年

月

日

一天中的哪个时间段(hour of day)

一周中的第几天

一年中的第几个星期

一年中的第几个季度

比如:

CUSTOMER ID	PRODUCT ID	PRICE	DATE TIME STAMP
1	5	5.99	2015–09–11 11:23
1	3	2.99	2015–01–12 17:50

```
day_of_week = lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S" ).w
eekday()
month = lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S" ).month
# please read docs on how week numbers are calculate
week_number = lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S" ).s
trftime('%V')
```

```
seasons = [0,0,1,1,1,2,2,2,3,3,3,0] #dec - feb is winter, then sprin
g, summer, fall etc
season = lambda x: seasons[(datetime.strptime(x, "%Y-%m-%d %H:%M:%S"
).month-1)]

# sleep: 12-5, 6-9: breakfast, 10-14: lunch, 14-17: dinner prep, 17-
21: dinner, 21-23: deserts!
times_of_day = [0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4,
4, 4, 4, 5, 5, 5 ]
time_of_day = lambda x: times_of_day[datetime.strptime(x, "%Y-%m-%d
%H:%M:%S").hour]
```

d.文本型

词袋(Bag of Words,BOW)

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

This model has many parameters, however the default values are quite reasonable (please see the reference documentation for the details):

```
>>> vectorizer = CountVectorizer(min_df=1)
>>> vectorizer
CountVectorizer(analyzer=...'word', binary=False, decode_error=...'strict',
        dtype=<... '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)
```

Let's use it to tokenize and count the word occurrences of a minimalistic corpus of text documents:

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<4x9 sparse matrix of type '<... 'numpy.int64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

N-Gram:词袋模型会完全丢失每个词的上下文信息,于是提出了n-gram语言模型.

```
>>> bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
...                                     token_pattern=r'\b\w+\b', min_df=1)
>>> analyze = bigram_vectorizer.build_analyzer()
>>> analyze('Bi-grams are cool!') == (
...     ['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool'])
True
```

The vocabulary extracted by this vectorizer is hence much bigger and can now resolve ambiguities encoded in local positioning patterns:

```
>>> X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
>>> X_2
...
array([[0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0],
       [0, 0, 1, 0, 0, 1, 1, 0, 0, 2, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0],
       [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1]]...)
```

In particular the interrogative form “Is this” is only present in the last document:

```
>>> feature_index = bigram_vectorizer.vocabulary_.get('is this')
>>> X_2[:, feature_index]
array([0, 0, 0, 1]...)
```

TF-IDF:是一种统计方法， 用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。 字词的重要性随着它在文件中出现的次数成正比增加， 但同时会随着它在语料库中出现的频率成反比下降.

TF(t) = (词t在当前文中出现次数) / (t在全部文档中出现次数)

IDF(t) = ln(总文档数/ 含t的文档数)

TF-IDF权重 = TF(t) * IDF(t)

Word2Vec:todo

e.统计特征

加减平均: 商品价格高于平均价格多少， 用户在某个品类下消费超过平均用户多少， 用户连续登录天数超过平均多少...

分位线： 商品属于售出商品价格的多少分位线处

次序型： 排在第几位

比例类： 电商中， 好/中/差评比例,你已超过全国百分之… 的同学

f.组合特征

简单组合特征:拼接型

user_id&&category: 10001&&女裙 10002&&男士牛仔

模型特征组合

用GBDT产出特征组合路径

组合特征和原始特征一起放进LR训练

g.特征选择

原因:

冗余： 部分特征的相关度太高了， 消耗计算性能。

噪声： 部分特征是对预测结果有负影响

特征选择 VS 降维:

前者只踢掉原本特征里和结果预测关系不大的， 后者做特征的计算组合构成新特征

SVD或者PCA确实也能解决一定的高维度问题

过滤式:

评估单个特征和target之间的相关程度， 排序留下Top相关的特征部分。

根据Pearson相关系数选择:

```
import numpy as np
from scipy.stats import pearsonr
np.random.seed(0)
size = 300
x = np.random.normal(0, 1, size)
print "Lower noise", pearsonr(x, x + np.random.normal(0, 1, size))
print "Higher noise", pearsonr(x, x + np.random.normal(0, 10, size))
```

```
Lower noise (0.71824836862138386, 7.3240173129992273e-49)
Higher noise (0.057964292079338148, 0.31700993885324746)
```

根据方差选择:

```
from sklearn.feature_selection import VarianceThreshold
VarianceThreshold(threshold=3).fit_transform(iris.data)
#方差选择法，返回值为特征选择后的数据
#参数threshold为方差的阈值
```

卡方检验:

经典的卡方检验是检验定性自变量对定性因变量的相关性。假设自变量有N种取值，因变量有M种取值，考虑自变量等于i且因变量等于j的样本频数的观察值与期望的差距，构建统计量:

$$\chi^2 = \sum \frac{(A-E)^2}{E}$$

这个统计量的含义简而言之就是自变量对因变量的相关性。用feature_selection库的SelectKBest类结合卡方检验来选择特征的代码如下:

- **SelectKBest** removes all but the k highest scoring features
- **SelectPercentile** removes all but a user-specified highest scoring percentage of features
- using common univariate statistical tests for each feature: false positive rate **SelectFpr**, false discovery rate **SelectFdr**, or family wise error **SelectFwe**.
- **GenericUnivariateSelect** allows to perform univariate feature selection with a configurable strategy. This allows to select the best univariate selection strategy with hyper-parameter search estimator.

For instance, we can perform a χ^2 test to the samples to retrieve only the two best features as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import chi2
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(chi2, k=2).fit_transform(X, y)
>>> X_new.shape
(150, 2)
```

SelectKBest---留下K个最相关的特征

SelectPercentile---留下某一比例数量(参数)的特征

互信息法:

经典的互信息也是评价定性自变量对定性因变量的相关性的，互信息计算公式如下:

$$I(X;Y) = \sum_{x \epsilon X} \sum_{y \epsilon Y} p(x,y) log \frac{p(x,y)}{p(x)p(y)}$$

了处理定量数据，最大信息系数法被提出，使用feature_selection库的SelectKBest类结合最大信息系数法来选择特征的代码如下:

```
from sklearn.feature_selection import SelectKBest
from minepy import MINE
def mic(x, y):
    m = MINE()
 m.compute_score(x, y)
```

```
 return (m.mic(), 0.5)
SelectKBest(lambda X, Y: array(map(lambda x:mic(x, Y), X.T)).T,
k=2).fit_transform(iris.data, iris.target)
```

缺点： 没有考虑到特征之间的关联作用， 可能把有用的关联特征误踢掉。

包裹型:

把特征选择看做一个特征子集搜索问题， 筛选各种特征子集， 用模型评估效果.典型的包裹型算法为 “递归特征删除算法” (recursive feature elimination algorithm)

比如,当选用的模型是LR时,具体步骤如下:

① 用全量特征跑一个模型

② 根据线性模型的系数(体现相关性)， 删掉5-10%的弱特征，观察准确率/auc的变化

③ 逐步进行， 直至准确率/auc出现大的下滑停止

包裹型Python包:

1.13.3. Recursive feature elimination

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), recursive feature elimination (`RFE`) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

`RFECV` performs RFE in a cross-validation loop to find the optimal number of features.

sklearn.feature_selection.RFE

class sklearn.feature_selection. **RFE** (*estimator, n_features_to_select=None, step=1, estimator_params=None, verbose=0*) [source]

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression

boston = load_boston()
X = boston["data"]
Y = boston["target"]
names = boston["feature_names"]

#use linear regression as the model
lr = LinearRegression()
#rank all features, i.e continue the elimination until the last one
rfe = RFE(lr, n_features_to_select=1)
rfe.fit(X,Y)

print "Features sorted by their rank:"
print sorted(zip(map(lambda x: round(x, 4), rfe.ranking_), names))
```

```
Features sorted by their rank:
[(1.0, 'NOX'), (2.0, 'RM'), (3.0, 'CHAS'), (4.0, 'PTRATIO'), (5.0, 'DIS'), (6.0,
'LSTAT'), (7.0, 'RAD'), (8.0, 'CRIM'), (9.0, 'INDUS'), (10.0, 'ZN'), (11.0,
'TAX'), (12.0, 'B'), (13.0, 'AGE')]
```

参数estimator为用来筛选特征的模型,n_feature_to_select为需要保留的最小特征数

嵌入型:

根据模型来分析特征的重要性（有别于上面的方式，是从生产的模型权重等） 。

最常见的方式为用正则化方式来做特征选择。

举个例子， 最早在电商用LR做CTR预估， 在3-5亿维的系数特征上用L1正则化的LR模型。 剩余2-3千万的feature， 意味着其他的feature重要度不够。

基于惩罚项的特征选择法:

嵌入型特征选择Python包

1.13.4.1. L1-based feature selection

Linear models penalized with the L1 norm have sparse solutions: many of their estimated coefficients are zero. When the goal is to reduce the dimensionality of the data to use with another classifier, they can be used along with `feature_selection.SelectFromModel` to select the non-zero coefficients. In particular, sparse estimators useful for this purpose are the `linear_model.Lasso` for regression, and of `linear_model.LogisticRegression` and `svm.LinearSVC` for classification:

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(X, y)
>>> model = SelectFromModel(lsvc, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 3)
```

L1惩罚项降维的原理在于保留多个对目标值具有同等相关性的特征中的一个，所以没选到的特征不代表不重要。故，可结合L2惩罚项来优化。具体操作为：若一个特征在L1中的权值为1，选择在L2中权值差别不大且在L1中权值为0的特征构成同类集合，将这一集合中的特征平分L1中的权值，故需要构建一个新的逻辑回归模型：

```
class LR(LogisticRegression):
    def __init__(self, threshold=0.01, dual=False, tol=1e-4, C=1.0,
                 fit_intercept=True, intercept_scaling=1, class_weight=None,
                 random_state=None, solver='liblinear', max_iter=100,
                 multi_class='ovr', verbose=0, warm_start=False, n_jobs=1):

        #权值相近的阈值
        self.threshold = threshold
        LogisticRegression.__init__(self, penalty='l1', dual=dual, tol=tol, C=C,
                 fit_intercept=fit_intercept, intercept_scaling=intercept_scaling, class_weight=class_weight,
                 random_state=random_state, solver=solver, max_iter=max_iter,
                 multi_class=multi_class, verbose=verbose, warm_start=warm_start, n_jobs=n_jobs)
        #使用同样的参数创建L2逻辑回归
        self.l2 = LogisticRegression(penalty='l2', dual=dual, tol=tol, C=C, fit_intercept=fit_intercept, intercept_scaling=intercept_scaling, class_weight =
class_weight, random_state=random_state, solver=solver, max_iter=max_iter, multi_class=multi_class, verbose=verbose, warm_start=warm_start, n_jobs=n_jobs)

    def fit(self, X, y, sample_weight=None):
        #训练L1逻辑回归
        super(LR, self).fit(X, y, sample_weight=sample_weight)
        self.coef_old_ = self.coef_.copy()
        #训练L2逻辑回归
        self.l2.fit(X, y, sample_weight=sample_weight)

        cntOfRow, cntOfCol = self.coef_.shape
        #权值系数矩阵的行数对应目标值的种类数目
        for i in range(cntOfRow):
            for j in range(cntOfCol):
                coef = self.coef_[i][j]
                #L1逻辑回归的权值系数不为0
                if coef != 0:
                    idx = [j]
                    #对应在L2逻辑回归中的权值系数
                    coef1 = self.l2.coef_[i][j]
                    for k in range(cntOfCol):
                        coef2 = self.l2.coef_[i][k]
                        #在L2逻辑回归中，权值系数之差小于设定的阈值，且在L1中对应的权值为0
                        if abs(coef1-coef2) < self.threshold and j != k and self.coef_[i][k] == 0:
                            idx.append(k)
                    #计算这一类特征的权值系数均值
                    mean = coef / len(idx)
                    self.coef_[i][idx] = mean
        return self
```

使用feature_selection库的SelectFromModel类结合带L1以及L2惩罚项的逻辑回归模型，来选择特征的代码如下：

```
from sklearn.feature_selection import SelectFromModel
#带L1和L2惩罚项的逻辑回归作为基模型的特征选择
#参数threshold为权值系数之差的阈值
SelectFromModel(LR(threshold=0.5, C=0.1)).fit_transform(iris.data, iris.target)
```

基于树模型的特征选择法：

树模型中GBDT也可用来作为基模型进行特征选择，使用feature_selection库的SelectFromModel类

结合GBDT模型，来选择特征的代码如下：

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import GradientBoostingClassifier
```

```
SelectFromModel(GradientBoostingClassifier()).fit_transform(iris.data, iris.target)
```

h.降维

当特征选择完成后，可以直接训练模型了，但是可能由于特征矩阵过大，导致计算量大，训练时间长的问题，因此降低特征矩阵维度也是必不可少的。常见的降维方法除了以上提到的基于L1惩罚项的模型以外，另外还有主成分分析法（PCA）和线性判别分析（LDA），线性判别分析本身也是一个分类模型。PCA和LDA有很多的相似点，其本质是要将原始的样本映射到维度更低的样本空间中，但是PCA和LDA的映射目标不一样：PCA是为了让映射后的样本具有最大的发散性；而LDA是为了让映射后的样本有最好的分类性能。所以说PCA是一种无监督的降维方法，而LDA是一种有监督的降维方法。

使用decomposition库的PCA类选择特征的代码如下：

```
from sklearn.decomposition import PCA
#主成分分析法，返回降维后的数据
#参数n_components为主成分数目
PCA(n_components=2).fit_transform(iris.data)
```

使用lda库的LDA类选择特征的代码如下：

```
from sklearn.lda import LDA
#线性判别分析法，返回降维后的数据
#参数n_components为降维后的维数
LDA(n_components=2).fit_transform(iris.data, iris.target)
```

i.数据变换

常见的数据变换有基于多项式的、基于指数函数的、基于对数函数的。4个特征，度为2的多项式转换公式如下：

$$(x'_1,x'_2,x'_3,x'_4,x'_5,x'_6,x'_7,x'_8,x'_8,x'_{10},x'_{11},x'_{12},x'_{13},x'_{14},x'_{15}) = (1,x_1,x_2,x_3,x_4,x_1^2,x_1*x_2,x_1*x_3,x_1*x_4,x_2^2,x_2*x_3,x_2*x_4,x_3^2,x_3*x_4,x_4^2)$$

```
from sklearn.preprocessing import PolynomialFeatures
PolynomialFeatures().fit_transform(iris.data) //#参数degree为度，默认值为2
```

基于单变元函数的数据变换可以使用一个统一的方式完成，使用preproccessing库的FunctionTransformer对数据进行对数函数转换的代码如下：

```
from numpy import log1p
from sklearn.preprocessing import FunctionTransformer
FunctionTransformer(log1p).fit_transform(iris.data) //自定义转换函数为对数函数的数据变换
                                                    第一个参数是单变元函数
```