Introduction

Kaggle 是目前最大的 Data Scientist 聚集地。很多公司会拿出自家的数据并提供奖金，在 Kaggle 上组织数据竞赛。我最近完成了第一次比赛，**在 2125 个参赛队伍中排名第 98 位（~ 5%）**。因为是第一次参赛，所以对这个成绩我已经很满意了。在 Kaggle 上一次比赛的结果除了排名以外，还会显示的就是 Prize Winner，10% 或是 25% 这三档。所以刚刚接触 Kaggle 的人很多都会以 25% 或是 10% 为目标。在本文中，我试图根据自己第一次比赛的经验和从其他 Kaggler 那里学到的知识，为刚刚听说 Kaggle 想要参赛的新手提供一些切实可行的冲刺 10% 的指导。

Kaggler 绝大多数都是用 Python 和 R 这两门语言的。因为我主要使用 Python，所以本文提到的例子都会根据 Python 来。不过 R 的用户应该也能不费力地了解到工具背后的思想。

首先简单介绍一些关于 Kaggle 比赛的知识：

- 不同比赛有不同的任务，分类、回归、推荐、排序等。比赛开始后训练集和测试集就会开放下载。
- 比赛通常持续 2 ~ 3 个月，每个队伍每天可以提交的次数有限，通常为 5 次。
- 比赛结束前一周是一个 Deadline，在这之后不能再组队，也不能再新加入比赛。所以**想要参加比赛请务必在这一 Deadline 之前有过至少一次有效的提交**。
- 一般情况下在提交后会立刻得到得分的反馈。不同比赛会采取不同的评分基准，可以在分数栏最上方看到使用的评分方法。
- 反馈的分数是基于测试集的一部分计算的，剩下的另一部分会被用于计算最终的结果。所以最后排名会变动。
- **LB** 指的就是在 Leaderboard 得到的分数，由上，有 **Public LB** 和 **Private LB** 之分。
- 自己做的 Cross Validation 得到的分数一般称为 **CV** 或是 **Local CV**。一般来说 **CV** 的结果比 **LB** 要可靠。
- 新手可以从比赛的 **Forum** 和 **Scripts** 中找到许多有用的经验和洞见。不要吝啬提问，Kaggler 都很热情。

那么就开始吧！

P.S. 本文假设读者对 Machine Learning 的基本概念和常见模型已经有一定了解。 Enjoy Reading!

General Approach

在这一节中我会讲述一次 Kaggle 比赛的大致流程。

Data Exploration

在这一步要做的基本就是 **EDA (Exploratory Data Analysis)**，也就是对数据进行探索性的分析，从而为之后的处理和建模提供必要的结论。

通常我们会用 **pandas** 来载入数据，并做一些简单的可视化来理解数据。

Visualization

通常来说 **matplotlib** 和 **seaborn** 提供的绘图功能就可以满足需求了。

比较常用的图表有：

- 查看目标变量的分布。当分布不平衡时，根据评分标准和具体模型的使用不同，可能会严重影响性能。
- 对 **Numerical Variable**，可以用 **Box Plot** 来直观地查看它的分布。
- 对于坐标类数据，可以用 **Scatter Plot** 来查看它们的分布趋势和是否有离群点的存在。
- 对于分类问题，将数据根据 Label 的不同着不同的颜色绘制出来，这对 Feature 的构造很有帮助。
- 绘制变量之间两两的分布和相关度图表。

这里**有一个在著名的 Iris 数据集上做了一系列可视化的例子，非常有启发性。**

Statistical Tests

我们可以对数据进行一些统计上的测试来验证一些假设的显著性。虽然大部分情况下靠可视化就能得到比较明确的结论，但有一些定量结果总是更理想的。不过，在实际数据中经常会遇到非 i.i.d. 的分布。所以要注意测试类型的的选择和对显著性的解释。

在某些比赛中，由于数据分布比较奇葩或是噪声过强，**Public LB** 的分数可能会跟 **Local CV** 的结果相去甚远。可以根据一些统计测试的结果来粗略地建立一个阈值，用来衡量一次分数的提高究竟是实质的提高还是由于数据的随机性导致的。

Data Preprocessing

大部分情况下，在构造 Feature 之前，我们需要对比赛提供的数据集进行一些处理。通常的步骤有：

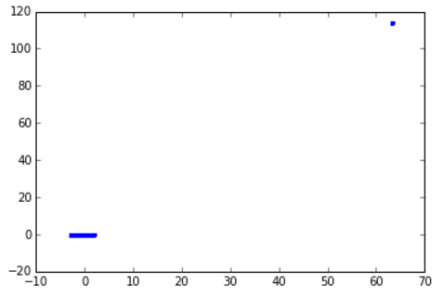
- 有时数据会分散在几个不同的文件中，需要 Join 起来。
- 处理 **Missing Data**。
- 处理 **Outlier**。
- 必要时转换某些 **Categorical Variable** 的表示方式。
- 有些 Float 变量可能是从未知的 Int 变量转换得到的，这个过程中发生精度损失会在数据中产生不必要的 **Noise**，即两个数值原本是相同的却在小数点后某一位开始有不同。这对 Model 可能会产生很负面的影响，需要设法去除或者减弱 Noise。

这一部分的处理策略多半依赖于在前一步中探索数据集所得到的结论以及创建的可视化图表。在实践中，我建议使用 **iPython Notebook** 进行对数据的操作，并熟练掌握常用的 pandas 函数。这样做的好处是可以随时得到结果的反馈和进行修改，也方便跟其他人进行交流（在 Data Science 中 Reproducible Results 是很重要的)。

下面给两个例子。

Outlier

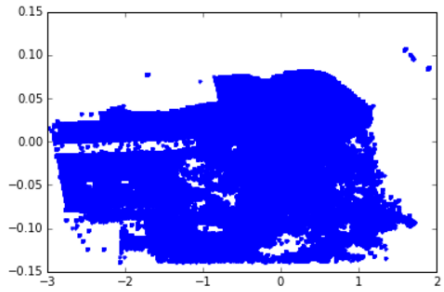
In [5]:
```
plt.plot(trainDF['X'], trainDF['Y'], '.')
plt.show()
```



We can see that there are some erroneous y values. Drop them.

In [5]:
```
trainDF = trainDF[abs(trainDF['Y'])<100]
```

In [7]:
```
plt.plot(trainDF['X'], trainDF['Y'], '.')
plt.show()
```



这是经过 Scaling 的坐标数据。可以发现右上角存在一些离群点，去除以后分布比较正常。

Dummy Variables

对于 Categorical Variable，常用的做法就是 One-hot encoding。即对这一变量创建一组新的伪变量，对应其所有可能的取值。这些变量中只有这条数据对应的取值为 1，其他都为 0。

如下，将原本有 7 种可能取值的 `Weekdays` 变量转换成 7 个 Dummy Variables。

In [31]:
```
trainDF.head(3)
```

Out[31]:

	DayOfWeek
262468	Saturday
217465	Sunday
223264	Friday

In [32]:
```
DayOfWeek_dummies = pd.get_dummies(trainDF['DayOfWeek'], prefix='DayOfWeek')
trainDF.drop(['DayOfWeek'], axis=1, inplace=True)
trainDF = trainDF.join(DayOfWeek_dummies)
```

In [34]:
```
trainDF.columns.tolist()
```

Out[34]:
```
['DayOfWeek_Friday',
 'DayOfWeek_Monday',
 'DayOfWeek_Saturday',
 'DayOfWeek_Sunday',
 'DayOfWeek_Thursday',
 'DayOfWeek_Tuesday',
 'DayOfWeek_Wednesday']
```

In [33]:
```
trainDF.head(3)
```

Out[33]:

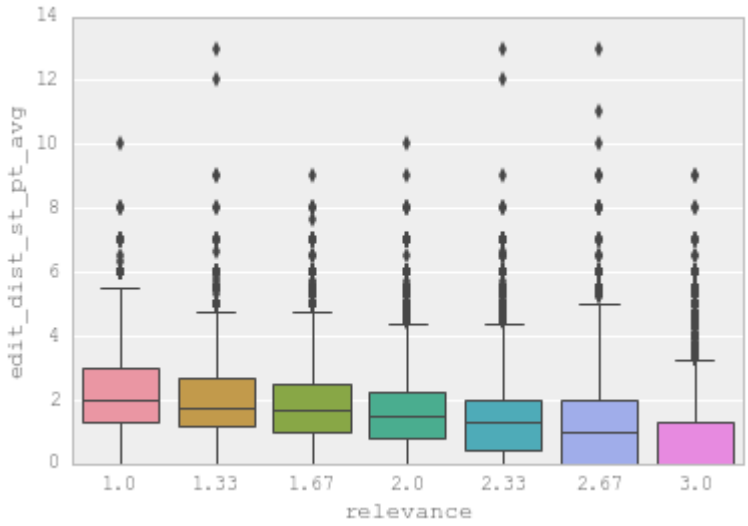
	DayOfWeek_Friday	DayOfWeek_Monday	DayOfWeek_Saturday	DayOfWeek_Sunday	DayOfWeek_Thu
262468	0	0	1	0	0
217465	0	0	0	1	0
223264	1	0	0	0	0

要注意，当变量可能取值的范围很大（比如一共有成百上千类）时，这种简单的方法就不太适用了。这时没有有一个普适的方法，但我会在下一小节描述其中一种。

Feature Engineering

有人总结 Kaggle 比赛是 **“Feature 为主，调参和 Ensemble 为辅”**，我觉得很有道理。Feature Engineering 能做到什么程度，取决于对数据领域的了解程度。比如在数据包含大量文本的比赛中，常用的 NLP 特征就是必须的。怎么构造有用的 Feature，是一个不断学习和提高的过程。

一般来说，**当一个变量从直觉上来说对所要完成的目标有帮助，就可以将其作为 Feature**。至于它是否有效，最简单的方式就是通过图表来直观感受。比如：



Feature Selection

总的来说，我们应该**生成尽量多的 Feature，相信 Model 能够挑出最有用的 Feature**。但有时先做一遍 Feature Selection 也能带来一些好处：

- Feature 越少，训练越快。
- 有些 Feature 之间可能存在线性关系，影响 Model 的性能。
- **通过挑选出最重要的 Feature，可以将它们之间进行各种运算和操作的结果作为新的 Feature，可能带来意外的提高。**

Feature Selection 最实用的方法也就是看 Random Forest 训练完以后得到的 **Feature Importance** 了。其他有一些更复杂的算法在理论上更加 Robust，但是缺乏实用高效的实现，比如这个。从原理上来讲，增加 Random Forest 中树的数量可以在一定程度上加强其对于 Noisy Data 的 Robustness。

看 Feature Importance 对于某些数据经过脱敏处理的比赛尤其重要。这可以免得你浪费大把时间在琢磨一个不重要的变量的意义上。

Feature Encoding

这里用一个例子来说明在一些情况下 Raw Feature 可能需要经过一些转换才能起到比较好的效果。

假设有一个 Categorical Variable 一共有几万个取值可能，那么创建 Dummy Variables 的方法就不可行了。这时一个比较好的方法是根据 Feature Importance 或是这些取值本身在数据中的出现频率，为最重要（比如说前 95% 的 Importance）那些取值（有很大可能只有几个或是十几个）创建 Dummy Variables，而所有其他取值都归到一个“其他”类里面。

Model Selection

准备好 Feature 以后，就可以开始选用一些常见的模型进行训练了。Kaggle 上最常用的模型基本都是基于树的模型：

- **Gradient Boosting**
- Random Forest
- Extra Randomized Trees

以下模型往往在性能上稍逊一筹，但是很适合作为 Ensemble 的 Base Model。这一点之后再详细解释。（当然，在跟图像有关的比赛中神经网络的重要性还是不能小觑的。）

- SVM
- Linear Regression
- Logistic Regression
- Neural Networks

以上这些模型基本都可以通过 **sklearn** 来使用。

当然，这里不能不提一下 **Xgboost**。**Gradient Boosting** 本身优秀的性能加上 **Xgboost** 高效的实现，使得它在 Kaggle 上广为使用。几乎每场比赛的获奖者都会用 **Xgboost** 作为最终 Model 的重要组成部分。在实战中，我们往往会以 Xgboost 为主来建立我们的模型并且验证 Feature 的有效性。顺带一提，**在 Windows 上安装 Xgboost 很容易遇到问题，目前已知最简单、成功率最高的方案可以参考我在这篇帖子中的描述**。

Model Training

在训练时，我们主要希望通过调整参数来得到一个性能不错的模型。一个模型往往有很多参数，但其中比较重要的一般不会太多。比如对 **sklearn** 的 `RandomForestClassifier` 来说，比较重要的就是随机森林中树的数量 `n_estimators` 以及在训练每棵树时最多选择的特征数量 `max_features`。所以**我们需要对自己使用的模型有足够的了解，知道每个参数对性能的影响是怎样的**。

通常我们会通过一个叫做 Grid Search 的过程来确定一组最佳的参数。其实这个过程说白了就是根据给定的参数候选对所有的组合进行暴力搜索。

```
param_grid = {'n_estimators': [300, 500], 'max_features': [10, 12, 14]}
model = grid_search.GridSearchCV(estimator=rfr, param_grid=param_grid,
n_jobs=1, cv=10, verbose=20, scoring=RMSE)
model.fit(X_train, y_train)
```

顺带一提，Random Forest 一般在 `max_features` 设为 Feature 数量的平方根附近得到最佳结果。

这里要重点讲一下 Xgboost 的调参。通常认为对它性能影响较大的参数有：

- `eta`：每次迭代完成后更新权重时的步长。越小训练越慢。
- `num_round`：总共迭代的次数。
- `subsample`：训练每棵树时用来训练的数据占全部的比例。用于防止 Overfitting。
- `colsample_bytree`：训练每棵树时用来训练的特征的比例，类似 `RandomForestClassifier` 的 `max_features`。
- `max_depth`：每棵树的最大深度限制。与 Random Forest 不同，**Gradient Boosting 如果不对深度加以限制，最终是会 Overfit 的**。
- `early_stopping_rounds`：用于控制在 Out Of Sample 的验证集上连续多少个迭代的分数都没有提高后就提前终止训练。用于防止 Overfitting。

一般的调参步骤是：

1. 将训练数据的一部分划出来作为验证集。
2. 先将 `eta` 设得比较高（比如 0.1），`num_round` 设为 300 ~ 500。
3. 用 Grid Search 对其他参数进行搜索
4. 逐步将 `eta` 降低，找到最佳值。
5. 以验证集为 watchlist，用找到的最佳参数组合重新在训练集上训练。注意观察算法的输出，看每次迭代后在验证集上分数的变化情况，从而得到最佳的 `early_stopping_rounds`。

```
 X_dtrain, X_deval, y_dtrain, y_deval =
1cross_validation.train_test_split(X_train, y_train, random_state=1026,
2test_size=0.3)
3dtrain = xgb.DMatrix(X_dtrain, y_dtrain)
4deval = xgb.DMatrix(X_deval, y_deval)
5watchlist = [(deval, 'eval')]
6params = {
7    'booster': 'gbtree',
8    'objective': 'reg:linear',
9    'subsample': 0.8,
10    'colsample_bytree': 0.85,
11    'eta': 0.05,
12    'max_depth': 7,
13    'seed': 2016,
14    'silent': 0,
15    'eval_metric': 'rmse'
16}
17clf = xgb.train(params, dtrain, 500, watchlist, early_stopping_rounds=50)
  pred = clf.predict(xgb.DMatrix(df_test))
```

最后要提一点，所有具有随机性的 Model 一般都会有一个 `seed` 或是 `random_state` 参数用于控制随机种子。得到一个好的 Model 后，在记录参数时务必也记录下这个值，从而能够在之后重现 Model。

Cross Validation

Cross Validation 是非常重要的一个环节。它让你知道你的 Model 有没有 Overfit，是不是真的能够 Generalize 到测试集上。在很多比赛中 **Public LB** 都会因为这样那样的原因而不可靠。当你改进了 Feature 或是 Model 得到了一个更高的 **CV** 结果，提交之后得到的 **LB** 结果却变差了，**一般认为这时应该相信 CV 的结果**。当然，最理想的情况是多种不同的 **CV** 方法得到的结果和 **LB** 同时提高，但这样的比赛并不是太多。

在数据的分布比较随机均衡的情况下，**5-Fold CV** 一般就足够了。如果不放心，可以提到 **10-Fold**。**但是 Fold 越多训练也就会越慢，需要根据实际情况进行取舍。**

很多时候简单的 **CV** 得到的分数会不大靠谱，Kaggle 上也有很多关于如何做 **CV** 的讨论。比如这个。但总的来说，靠谱的 **CV** 方法是 Case By Case 的，需要在实际比赛中进行尝试和学习，这里就不再（也不能）叙述了。

Ensemble Generation

Ensemble Learning 是指将多个不同的 Base Model 组合成一个 Ensemble Model 的方法。它可以**同时降低最终模型的 Bias 和 Variance**（证明可以参考这篇论文，我最近在研究类似的理论，可能之后会写新文章详述），

从而在提高分数的同时又降低 Overfitting 的风险。在现在的 Kaggle 比赛中要不用 Ensemble 就拿到奖金几乎是不可能的。

常见的 Ensemble 方法有这么几种：

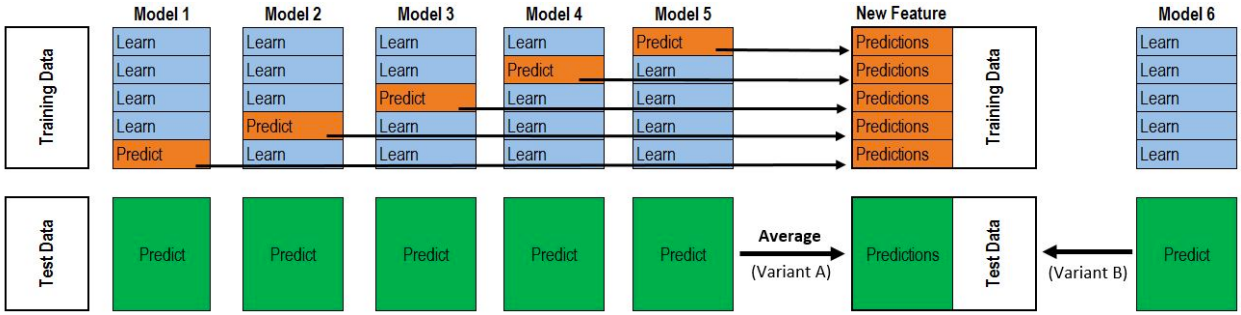
- Bagging：使用训练数据的不同随机子集来训练每个 Base Model，最后进行每个 Base Model 权重相同的 Vote。也即 Random Forest 的原理。
- Boosting：迭代地训练 Base Model，每次根据上一个迭代中预测错误的情况修改训练样本的权重。也即 Gradient Boosting 的原理。比 Bagging 效果好，但更容易 Overfit。
- Blending：用不相交的数据训练不同的 Base Model，将它们的输出取（加权）平均。实现简单，但对训练数据利用少了。
- Stacking：接下来会详细介绍。

从理论上讲，Ensemble 要成功，有两个要素：

- **Base Model 之间的相关性要尽可能的小。**这就是为什么非 Tree-based Model 往往表现不是最好但还是要将它们包括在 Ensemble 里面的原因。Ensemble 的 Diversity 越大，最终 Model 的 Bias 就越低。
- **Base Model 之间的性能表现不能差距太大。**这其实是一个 **Trade-off**，在实际中很有可能表现相近的 Model 只有寥寥几个而且它们之间相关性还不低。但是实践告诉我们即使在这种情况下 Ensemble 还是能大幅提高成绩。

Stacking

相比 Blending，Stacking 能更好地利用训练数据。以 5-Fold Stacking 为例，它的基本原理如图所示：



整个过程很像 Cross Validation。首先将训练数据分为 5 份，接下来一共 5 个迭代，每次迭代时，将 4 份数据作为 Training Set 对每个 Base Model 进行训练，然后在剩下一份 Hold-out Set 上进行预测。**同时也要将其在测试数据上的预测保存下来。**这样，每个 Base Model 在每次迭代时会对训练数据的其中 1 份做出预测，对测试数据的全部做出预测。5 个迭代都完成以后我们就获得了一个 `#训练数据行数 x #Base Model 数量` 的矩阵，这个矩阵接下来就作为第二层的 Model 的训练数据。当第二层的 Model 训练完以后，将之前保存的 Base Model 对测试数据的预测（**因为每个 Base Model 被训练了 5 次，对测试数据的全体做了 5 次预测，所以对这 5 次求一个平均值，从而得到一个形状与第二层训练数据相同的矩阵**）拿出来让它进行预测，就得到最后的输出。

这里给出我的实现代码：

```python
class Ensemble(object):
    def __init__(self, n_folds, stacker, base_models):
        self.n_folds = n_folds
        self.stacker = stacker
        self.base_models = base_models

    def fit_predict(self, X, y, T):
        X = np.array(X)
        y = np.array(y)
        T = np.array(T)

        folds = list(KFold(len(y), n_folds=self.n_folds, shuffle=True,
random_state=2016))

        S_train = np.zeros((X.shape[0], len(self.base_models)))
        S_test = np.zeros((T.shape[0], len(self.base_models)))

        for i, clf in enumerate(self.base_models):
            S_test_i = np.zeros((T.shape[0], len(folds)))

            for j, (train_idx, test_idx) in enumerate(folds):
                X_train = X[train_idx]
                y_train = y[train_idx]
                X_holdout = X[test_idx]
                # y_holdout = y[test_idx]
                clf.fit(X_train, y_train)
                y_pred = clf.predict(X_holdout)[:]
                S_train[test_idx, i] = y_pred
                S_test_i[:, j] = clf.predict(T)[:]

            S_test[:, i] = S_test_i.mean(1)

        self.stacker.fit(S_train, y)
        y_pred = self.stacker.predict(S_test)[:]
        return y_pred
```

获奖选手往往会使用比这复杂得多的 Ensemble，会出现三层、四层甚至五层，不同的层数之间有各种交互，还有将经过不同的 Preprocessing 和不同的 Feature Engineering 的数据用 Ensemble 组合起来的做法。但对于新手来说，稳稳当当地实现一个正确的 5-Fold Stacking 已经足够了。

*Pipeline

可以看出 Kaggle 比赛的 Workflow 还是比较复杂的。尤其是 Model Selection 和 Ensemble。理想情况下，我们需要搭建一个高自动化的 Pipeline，它可以做到：

- **模块化 Feature Transform**，只需写很少的代码就能将新的 Feature 更新到训练集中。
- **自动化 Grid Search**，只要预先设定好使用的 Model 和参数的候选，就能自动搜索并记录最佳的 Model。
- **自动化 Ensemble Generation**，每个一段时间将现有最好的 K 个 Model 拿来做 Ensemble。

对新手来说，第一点可能意义还不是太大，因为 Feature 的数量总是人脑管理的过来的；第三点问题也不大，因为往往就是在最后做几次 Ensemble。但是第二点还是很有意义的，手工记录每个 Model 的表现不仅浪费时间而且容易产生混乱。

Crowdflower Search Results Relevance 的第一名获得者 Chenglong Chen 将他在比赛中使用的 Pipeline 公开了，非常具有参考和借鉴意义。只不过看懂他的代码并将其中的逻辑抽离出来搭建这样一个框架，还是比较困难的一件事。可能在参加过几次比赛以后专门抽时间出来做会比较好。

Home Depot Search Relevance

在这一节中我会具体分享我在 Home Depot Search Relevance 比赛中是怎么做的，以及比赛结束后从排名靠前的队伍那边学到的做法。

首先简单介绍这个比赛。Task 是**判断用户搜索的关键词和网站返回的结果之间的相关度有多高**。相关度是由 3 个人类打分取平均得到的，每个人可能打 1 ~ 3 分，所以这是一个回归问题。数据中包含用户的搜索词，返回的产品的标题和介绍，以及产品相关的一些属性比如品牌、尺寸、颜色等。使用的评分基准是 RMSE。

这个比赛非常像 Crowdflower Search Results Relevance 那场比赛。不过那边用的评分基准是 Quadratic Weighted Kappa，把 1 误判成 4 的惩罚会比把 1 判成 2 的惩罚大得多，所以在最后 Decode Prediction 的时候会更麻烦一点。除此以外那次比赛没有提供产品的属性。

EDA

由于加入比赛比较晚，当时已经有相当不错的 EDA 了。尤其是这个。从中我得到的启发有：

- 同一个搜索词/产品都出现了多次，**数据分布显然不 i.i.d.**。
- 文本之间的相似度很有用。
- 产品中有相当大一部分缺失属性，要考虑这会不会使得从属性中得到的 Feature 反而难以利用。
- 产品的 ID 对预测相关度很有帮助，但是考虑到训练集和测试集之间的重叠度并不太高，利用它会不会导致 Overfitting？

Preprocessing

这次比赛中我的 Preprocessing 和 Feature Engineering 的具体做法都可以在这里看到。我只简单总结一下和指出重要的点。

1. 利用 Forum 上的 Typo Dictionary 修正搜索词中的错误。
2. 统计属性的出现次数，将其中出现次数多又容易利用的记录下来。
3. 将训练集和测试集合并，并与产品描述和属性 Join 起来。这是考虑到后面有一系列操作，如果不合并的话就要重复写两次了。
4. 对所有文本能做 **Stemming** 和 **Tokenizing**，同时手工做了一部分**格式统一化（比如涉及到数字和单位的）**和**同义词替换**。

Feature

- *Attribute Features
    - 是否包含某个特定的属性（品牌、尺寸、颜色、重量、内用/外用、是否有能源之星认证等）
    - 这个特定的属性是否匹配
- Meta Features
    - 各个文本域的长度
    - 是否包含属性域

    - 品牌（将所有的品牌做数值离散化）
    - 产品 ID
- 简单匹配
    - 搜索词是否在产品标题、产品介绍或是产品属性中出现
    - 搜索词在产品标题、产品介绍或是产品属性中出现的数量和比例
    - *搜索词中的第 i 个词是否在产品标题、产品介绍或是产品属性中出现
- 搜索词和产品标题、产品介绍以及产品属性之间的文本相似度
    - BOW Cosine Similairty
    - TF-IDF Cosine Similarity
    - Jaccard Similarity
    - *Edit Distance
    - Word2Vec Distance（由于效果不好，最后没有使用，但似乎是因为用的不对）
- **Latent Semantic Indexing：通过将 BOW/TF-IDF Vectorization 得到的矩阵进行 SVD 分解，我们可以得到不同搜索词/产品组合的 Latent 标识。这个 Feature 使得 Model 能够在一定程度上对不同的组合做出区别，从而解决某些产品缺失某些 Feature 的问题。**

值得一提的是，上面打了 * 的 Feature 都是我在最后一批加上去的。问题是，使用这批 Feature 训练得到的 Model 反而比之前的要差，而且还差不少。我一开始是以为因为 Feature 的数量变多了所以一些参数需要重新调优，但在浪费了很多时间做 Grid Search 以后却发现还是没法超过之前的分数。这可能就是之前提到的 Feature 之间的相互作用导致的问题。当时我设想过一个看到过好几次的解决方案，就是**将使用不同版本 Feature 的 Model 通过 Ensemble 组合起来**。但最终因为时间关系没有实现。事实上排名靠前的队伍分享的解法里面基本都提到了将不同的 Preprocessing 和 Feature Engineering 做 Ensemble 是获胜的关键。

Model

我一开始用的是 `RandomForestRegressor`，后来在 Windows 上折腾 **Xgboost** 成功了就开始用 `XGBRegressor`。**XGB** 的优势非常明显，同样的数据它只需要不到一半的时间就能跑完，节约了很多时间。

比赛中后期我基本上就是一边台式机上跑 **Grid Search**，一边在笔记本上继续研究 Feature。

这次比赛数据分布很不独立，所以期间多次遇到改进的 Feature 或是 **Grid Search** 新得到的参数训练出来的模型反而 **LB** 分数下降了。由于被很多前辈教导过要相信自己的 **CV**，我的决定是将 5-Fold 提到 10-Fold，然后以 **CV** 为标准继续前进。

Ensemble

最终我的 Ensemble 的 Base Model 有以下四个：

- `RandomForestRegressor`
- `ExtraTreesRegressor`
- `GradientBoostingRegressor`
- `XGBRegressor`

第二层的 Model 还是用的 **XGB**。

因为 **Base Model** 之间的相关都都太高了（最低的一对也有 0.9），我原本还想引入使用 `gblinear` 的 `XGBRegressor` 以及 `SVR`，但前者的 RMSE 比其他几个 Model 高了 0.02（这在 **LB** 上有几百名的差距），而后者的训练实在太慢了。最后还是只用了这四个。

值得一提的是，在开始做 Stacking 以后，我的 CV 和 LB 成绩的提高就是完全同步的了。

在比赛最后两天，因为身心疲惫加上想不到还能有什么显著的改进，我做了一件事情：用 20 个不同的随机种子来生成 Ensemble，最后取 **Weighted Average**。这个其实算是一种变相的 Bagging。**其意义在于按我实现 Stacking 的方式，我在训练 Base Model 时只用了 80% 的训练数据，而训练第二层的 Model 时用了 100% 的数据，这在一定程度上增大了 Overfitting 的风险。而每次更改随机种子可以确保每次用的是不同的 80%，这样在多次训练取平均以后就相当于逼近了使用 100% 数据的效果。**这给我带来了大约 `0.0004` 的提高，也很难受说是真的有效还是随机性了。

比赛结束后我发现我最好的单个 Model 在 **Private LB** 上的得分是 `0.46378`，而最终 Stacking 的得分是 `0.45849`。这是 174 名和 98 名的差距。也就是说，我单靠 Feature 和调参进到了 前 10%，而 **Stacking** 使我进入了前 5%。

Lessons Learned

比赛结束后一些队伍分享了他们的解法，从中我学到了一些我没有做或是做的不够好的地方：

- 产品标题的组织方式是有 Pattern 的，比如一个产品是否带有某附件一定会用 `With/Without XXX` 的格式放在标题最后。
- 使用**外部数据**，比如 WordNet，Reddit 评论数据集等来训练同义词和上位词（在一定程度上替代 Word2Vec）词典。
- 基于**字母**而不是单词的 NLP Feature。这一点我让我十分费解，但请教以后发现非常有道理。举例说，排名第三的队伍在计算匹配度时，将搜索词和内容中相匹配的单词的长度也考虑进去了。这是因为他们发现**越长的单词约具体，所以越容易被用户认为相关度高**。此外他们还使用了逐字符的序列比较（`difflib.SequenceMatcher`），因为**这个相似度能够衡量视觉上的相似度**。像这样的 Feature 的确不是每个人都能想到的。
- 标注单词的词性，找出**中心词**，计算基于中心词的各种匹配度和距离。这一点我想到了，但没有时间尝试。
- 将产品标题/介绍中 TF-IDF 最高的一些 Trigram 拿出来，计算搜索词中出现在这些 Trigram 中的比例；反过来以搜索词为基底也做一遍。这相当于是从另一个角度抽取了一些 Latent 标识。
- 一些新颖的距离尺度，比如 Word Movers Distance
- 除了 SVD 以外还可以用上 NMF。
- **最重要的 Feature 之间的 Pairwise Polynomial Interaction**。
- **针对数据不 i.i.d. 的问题，在 CV 时手动构造测试集与验证集之间产品 ID 不重叠和重叠的两种不同分割，并以与实际训练集/测试集的分割相同的比例来做 CV 以逼近 LB 的得分分布**。

至于 Ensemble 的方法，我暂时还没有办法学到什么，因为自己只有最简单的 Stacking 经验。

Summary

Takeaways

1. 比较早的时候就开始做 Ensemble 是对的，这次比赛到倒数第三天我还在纠结 Feature。
2. 很有必要搭建一个 Pipeline，至少要能够自动训练并记录最佳参数。
3. Feature 为王。我花在 Feature 上的时间还是太少。
4. 可能的话，多花点时间去手动查看原始数据中的 Pattern。

Issues Raised

我认为在这次比赛中遇到的一些问题是很有研究价值的：

1. 在数据分布并不 i.i.d. 甚至有 Dependency 时如何做靠谱的 **CV**。
2. 如何量化 Ensemble 中 **Diversity vs. Accuracy** 的 Trade-off。
3. 如何处理 Feature 之间互相影响导致性能反而下降。

Beginner Tips

给新手的一些建议：

1. 选择一个感兴趣的比赛。**如果你对相关领域原本就有一些洞见那就更理想了。**
2. 根据我描述的方法开始探索、理解数据并进行建模。
3. 通过 Forum 和 Scripts 学习其他人对数据的理解和构建 Feature 的方式。
4. **如果之前有过类似的比赛，可以去找当时获奖者的 Interview 和 Blog Post 作为参考，往往很有用。**
5. 在得到一个比较不错的 **LB** 分数（比如已经接近前 10%）以后可以开始尝试做 Ensemble。
6. 如果觉得自己有希望拿到奖金，开始找人组队吧！
7. **到比赛结束为止要绷紧一口气不能断，尽量每天做一些新尝试。**
8. 比赛结束后学习排名靠前的队伍的方法，思考自己这次比赛中的不足和发现的问题，**可能的话再花点时间将学到的新东西用实验进行确认，为下一次比赛做准备**。
9. 好好休息！

Reference

1. Beating Kaggle the Easy Way - Dong Ying
2. Solution for Prudential Life Insurance Assessment - Nutastray
3. Search Results Relevance Winner’s Interview: 1st place, Chenglong Chen