

Binary Classification via a Reinforcement Learner

Joshua Cook

Abstract

The purpose of this project is to solve a Kaggle competition using manually constructed neural networks and reinforcement learning techniques. The [competition](#) in question is sponsored by Red Hat. Given situational (an “action” data set) and customer (a “people” data set) information, the goal is to predict customer behavior for a given action. Customer behavior is a binary classification; customers either take an action or they do not. This project will use these two data sources and neural network/reinforcement learning techniques to prepare an algorithm capable of predicting outcomes against a third situational (a “test action” data set) source. The infrastructure designed and built for this project is informed by and informs the work, [the Containerized Jupyter Platform](#). This work is accompanied by a set of [Jupyter notebooks](#) and a `docker-compose.yml` file that can be run in order to validate all information here presented. The data set as provided has been scrubbed of all information. All attributes have generic names like `attribute_1`, `attribute_2`, etc, all characteristics are `characteristic_1`, `characteristic_2`, etc, and all outcomes are `outcome_1`, `outcome_2`, etc. As such the competition presents an interesting challenge, in which domain knowledge is completely useless. The competition is in essence a “pure machine learning problem.”

Contents

1	Definition	4
1.1	Problem Statement	4
1.2	Approach	5
1.3	Metrics	5
1.3.1	Infrastructure	7
2	Preliminary Data Analysis	8
2.1	Connecting to PostgreSQL	8
2.2	Data Exploration	9
2.3	Seeding the Database	9
2.4	Basic PostgreSQL Descriptors	10
2.4.1	Descriptor for database	10
2.4.2	Descriptor for action table	10
2.4.3	Descriptor for people table	11
2.5	Define the Basic Structure	12
2.5.1	Number of Rows in database tables	12
2.5.2	Number of Columns per Data Type	12
2.6	Identify Unique Labels	12
2.6.1	Number of Unique Labels for people	12
2.6.2	Number of Unique Labels for action	13
2.7	Run Aggregates on Columns	13
2.8	Identify Duplicate Records	14
2.9	Search for NULL Data	15
2.10	Exploratory Visualiztion	16

3	Algorithms and Techniques	21
3.1	One-Hot Encoding	21
3.1.1	One-Hot Encoding Example	22
3.2	Linear Classification via Neural Network	22
3.2.1	Score Function	23
3.2.2	Loss Function	23
3.2.3	Regularization Penalty	24
3.2.4	Final Loss Function	24
3.3	Optimization	24
3.3.1	Generate a Random Weights Matrix	24
3.3.2	Randomly guessing	25
3.3.3	Random Local Search	25
3.3.4	Gradient Descent	25
3.4	Benchmark	25
3.4.1	Confusion Matrix	26
3.4.2	Accuracy	26
3.4.3	F1 Score	26
4	Free-Form Visualization	28
4.1	Visualizing the Loss Function	28
5	Data Preprocessing	31
5.1	CSV Manipulation	31
5.1.1	<code>act_train.csv</code>	31
5.1.2	<code>act_test.csv</code>	31
5.1.3	All Sets	31
5.2	One-Hot Encoding	32
6	Implementation	33
6.1	Steps to Implementation	33
6.2	Seed a PostgreSQL database with the three csv files.	33
6.3	One-Hot Encode the data and store the one-hot encoded vector as an array in the <code>action</code> table	33

6.4	Prediction on Random Weights Matrix With No Training	34
6.4.1	Second Pass with a Random Matrix	35
6.4.2	Third Pass with a Random Matrix	36
6.5	Results from Training on a Random Matrix	36
7	Refinement	37
7.1	Learning via Random Search	37
7.1.1	Establish State of Training Session	38
7.1.2	Assess Results	38
7.2	Learning via Random Local Search	39
7.2.1	Establish State of Training Session	39
7.2.2	Assess Results	39
7.3	Learning via Gradient Descent	40
7.3.1	Establish State of Training Session	41
7.3.2	Assess Results	41
7.4	Recover from Curse of Dimensionality	42
7.5	Rewrite One-Hot to Action Table	42
7.6	Learning through a multi-layer Neural Network	42
8	Results	44
8.1	Final Model	44
8.2	Benchmark Comparison	44
9	Conclusion	45
9.1	Summary of Project	45
9.2	Steps for Improvment	46
9.3	Reflection	46

Chapter 1

Definition

Please refer to notebook [1 Definition](#).

1.1 Problem Statement

In this Kaggle competition, Red Hat seeks an optimal algorithm for using information about a given action and information about a given customer to predict the customer's behavior with regard to that action. A completed product will take the form of a csv with two items per row - an `action_id` from the test set, and a predicted outcome from the set 0, 1.

The following is a sample of the required format for a solution submission:

```
$ head data/sample_submission.csv

activity_id,outcome
act1_1,0
act1_100006,0
act1_100050,0
act1_100065,0
act1_100068,0
act1_100100,0
```

Data is provided in the form of three separate data sets encoded as CSV:

- `people.csv`
- `act_train.csv`
- `act_test.csv`.

We will store our data in two tables in a PostgreSQL Database. The `action` (`act_train.csv`) table makes reference to the `people` (`people.csv`) table. Beyond this, the sets have been scrubbed of any domain specific knowledge. Rather attributes are referred to generically as `char_1`, `char_2`, etc. As such the competition presents an interesting challenge, in which domain knowledge is completely useless. The competition is in essence a “pure machine learning problem.”

1.2 Approach

We take the following approach to completing this task:

1. Seed a PostgreSQL database with the three csv files.
2. One-Hot Encode the data and store the one-hot encoded vector as an array in the `action` table
3. Train and Assess a Series of Learners
 - try learning via random search
 - try learning via random local search
 - try learning via stochastic gradient descent
 - try learning through a multi-layer neural network

Note that while the Kaggle Challenge includes a set of test-data, for the purposes of this study we will be holding a separate test set aside that we are able to run our own local accuracy metrics. At the time of this writing, the competition is closed to new submissions.

To prevent overfitting, we will include a regularization penalty in our model.

1.3 Metrics

We will look at three different properties of our test set in order to measure the success of our learner:

1. a confusion matrix
2. accuracy
3. F1 Score

Our task is one of classification. The confusion matrix is specifically designed to show the success of a classifier by showing the number of times it correctly and incorrectly identified each target class. The accuracy aims to capture the success of the classifier by calculating a normalized count of the number of successful prediction. The F1 score is a third metric for classification which can be thought of as a weighted average of the precision and recall. The confusion matrix will provide a handy visual for examining our learners, whereas the accuracy and the F1 score will provide a numerical metric which can be optimized.

We will assess the learner against the test set throughout the training process as a way of assessing the development of our learner. However, the results of the development of the assessment will not be used for training and can thus be used repeatedly as an impartial measure of progress.

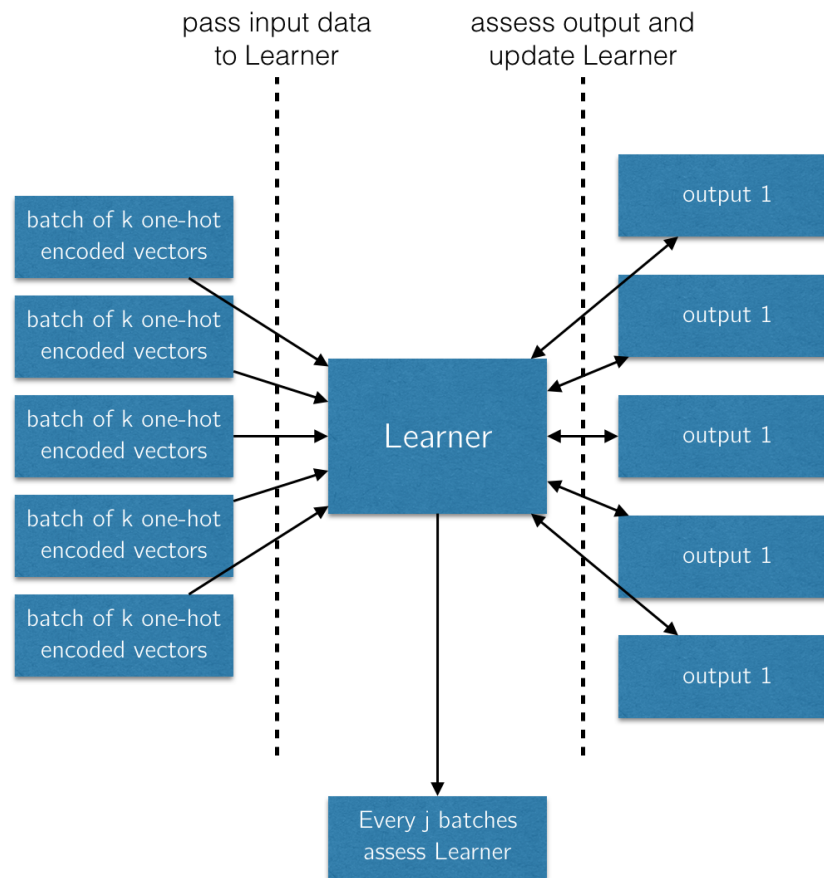


Figure 1.1: Learner Training and Assessment

1.3.1 Infrastructure

We have designed a special infrastructure geared toward a “back-end”/server-side implementation of our processes. This system uses Jupyter notebooks as its main interface, though it is possible to interface with the system via the terminal. Additionally, a browser-based control panel exists for tracking the progress of our workers. We use two data management systems, a PostgreSQL database and Redis. Finally, we have a worker layer of n scalable worker cpus built using Python’s `rq` framework.

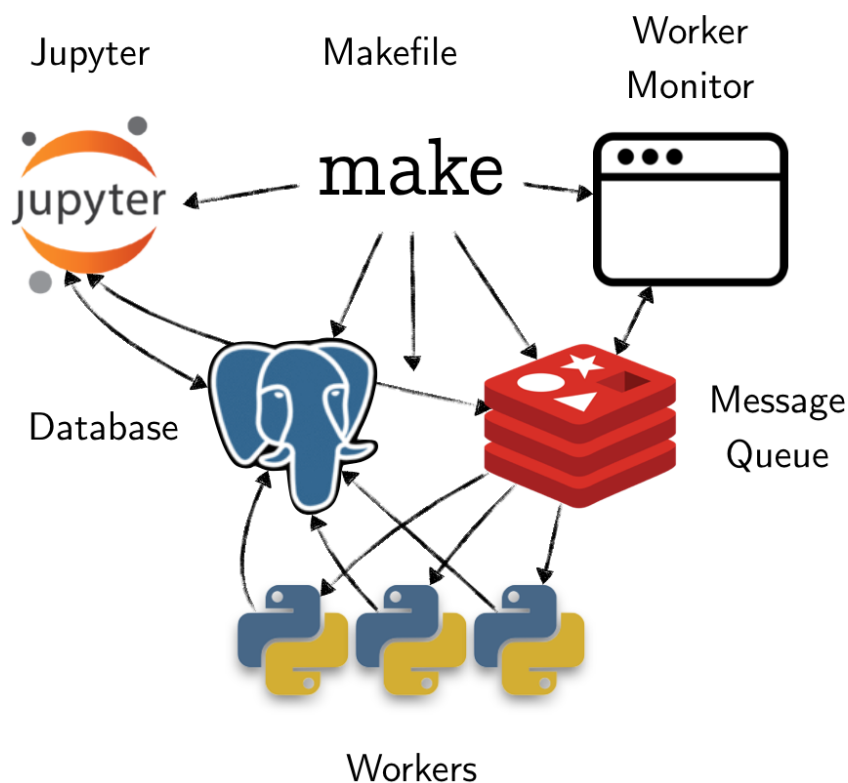


Figure 1.2: Infrastructure

Chapter 2

Preliminary Data Analysis

2.1 Connecting to PostgreSQL

Please refer to notebook [2.01 Preliminary Data Analysis - Connecting to PostgreSQL](#).

We store all included data in a PostgreSQL database. By and large, we access this database using the [psycpg2](#) library. Here, we make use of a development pattern we will use throughout the project in which more complicated are abstracted into modules in [lib](#). Here, we import `connect_to_postgres` from [lib/helpers/database_helper.py](#) and use it to connect to our database. Then, we run a simple query on the database, verifying that all is functioning well.

```
>>> from os import chdir; chdir('../')
>>> from lib.helpers.database_helper import connect_to_postgres
>>> conn, cur = connect_to_postgres()
>>> cur.execute("SELECT COUNT(*) FROM people"); print(cur.fetchone())
>>> cur.execute("SELECT COUNT(*) FROM action"); print(cur.fetchone())
(189118,)
(2695978,)
>>> conn.close()
```

2.2 Data Exploration

The data to be used here consists of three datasets:

- `people.csv` [sample](#)
- `act_train.csv` [sample](#)
- `act_test.csv` [sample](#)

We will do the following to analyze the datasets.

1. seeding the database
2. basic postgres descriptor (`\d+`)
3. define the basic structure - rows, columns, data types
4. identify unique labels for each column and the counts for each label
5. run aggregates on columns - mean, median, max, min
6. identify duplicate records, if they exist
7. search for NULL data
8. create histograms of data

2.3 Seeding the Database

This is handled during the building of the Docker image for our PostgreSQL database and is written into our database [Dockerfile](#).

In order to run the commands in this `Dockerfile` we use the `docker-compose` tool to build our image.

```
$ docker-compose build
```

During the building of the image, any `.sql` or `.sh` files located in `/docker-entrypoint-initdb.d` will be executed. We have defined the tables we will be using in the `tables.sql` file. The structure will be shown in a moment when we run the postgres descriptors. The full structure can be viewed in the seeding file [here](#). This functionality is part of the PostgreSQL public Docker image.

2.4 Basic PostgreSQL Descriptors

Having built and run our images, we now have a running PostgreSQL database that has been seeded with our csv data.

2.4.1 Descriptor for database

We use the PostgreSQL descriptor command to display basic attributes of our database.

```
postgres=## \d+
```

List of relations					
Schema	Name	Type	Owner	Size	Description
public	action	table	postgres	235 MB	
public	people	table	postgres	30 MB	

2.4.2 Descriptor for action table

We can repeat the same for a particular table. The tables have been trimmed so as not to show columns of repeating type.

```
postgres=## \d+ action
```

Table "public.action"	
Column	Type
people_id	text
act_id	text
act_date	timestamp without time zone
act_category	text
act_char_1	text
...	
act_char_10	text
act_outcome	boolean

Indexes:

"action_pkey" PRIMARY KEY, btree (act_id)

Foreign-key constraints:

"action_people_id_fkey" FOREIGN KEY (people_id) REFERENCES people(people_id)

2.4.3 Descriptor for people table

postgres=## \d+ people

Table "public.people"		
Column	Type	Modifiers
people_id	text	not null
ppl_char_1	text	
ppl_group_1	text	
ppl_char_2	text	
ppl_date	timestamp without time zone	
ppl_char_3	text	
...		
ppl_char_9	text	
ppl_char_10	boolean	
ppl_char_11	boolean	
ppl_char_12	boolean	
...		
ppl_char_37	boolean	
ppl_char_38	real	

Indexes:

"people_pkey" PRIMARY KEY, btree (people_id)

Referenced by:

TABLE "action" CONSTRAINT "action_people_id_fkey"

FOREIGN KEY (people_id) REFERENCES people(people_id)

2.5 Define the Basic Structure

Please refer to notebook [2.05 Preliminary Data Analysis - Define the Basic Structure](#).

The number of rows in a set can be identified by a query using the `COUNT()` function. Our test and training sets can be identified by the fact that the test set has `NULL` values in the `act_outcome` column.

2.5.1 Number of Rows in database tables

database	number of rows	number of training rows
people	189118	N/A
action	2695978	498687

2.5.2 Number of Columns per Data Type

database	text	boolean	timestamp	real
people	11	28	1	0
action	13	1	1	1

2.6 Identify Unique Labels

2.6.1 Number of Unique Labels for people

label	unique
people_id	189118
ppl_group_1	34224
ppl_date	1196
ppl_char_1	2
ppl_char_2	3
ppl_char_3	43
ppl_char_4	25
ppl_char_5	9
ppl_char_6	7
ppl_char_7	25
ppl_char_8	8
ppl_char_9	9

Additionally we do not show the final group of columns for the following reasons. `ppl_char_10` through `ppl_char_37` are boolean and have only two labels - `TRUE` and `FALSE`.

`ppl_char_38` is a continuous valued column.

2.6.2 Number of Unique Labels for action

Again we first show columns that have too many labels. However, upon second consideration we should use the column `act_category`.

label	unique
act_id	2695978
act_date	411
act_category	7
act_char_1	51
act_char_2	32
act_char_3	11
act_char_4	7
act_char_5	7
act_char_6	5
act_char_7	8
act_char_8	18
act_char_9	19
act_char_10	6969

We do not show the outcome `act_outcome` because it is boolean.

2.7 Run Aggregates on Columns

Next we take the average of our boolean columns. Note that all of them skew to the negation, most of them heavily so. The only exception is `act_outcome` which, while still toward the negation, is closer to the middle.

label	mean
ppl_char_10	(0.2509)
ppl_char_11	(0.2155)
ppl_char_12	(0.2403)
ppl_char_13	(0.3651)
ppl_char_14	(0.2598)
ppl_char_15	(0.2695)
ppl_char_16	(0.2821)

label	mean
ppl_char_17	(0.2920)
ppl_char_18	(0.1876)
ppl_char_19	(0.2847)
ppl_char_20	(0.2291)
ppl_char_21	(0.2850)
ppl_char_22	(0.2911)
ppl_char_23	(0.2985)
ppl_char_24	(0.1904)
ppl_char_25	(0.3278)
ppl_char_26	(0.1670)
ppl_char_27	(0.2381)
ppl_char_28	(0.2889)
ppl_char_29	(0.1683)
ppl_char_30	(0.2069)
ppl_char_31	(0.2786)
ppl_char_32	(0.2849)
ppl_char_33	(0.2178)
ppl_char_34	(0.3565)
ppl_char_35	(0.2103)
ppl_char_36	(0.3437)
ppl_char_37	(0.2855)
act_outcome	(0.4440)

Then we take the average, maximum, and minimum of the single real-valued column.

```
SELECT AVG(ppl_char_38), MAX(ppl_char_38), MIN(ppl_char_38) FROM people;
      avg      | max | min
-----+-----+-----
50.3273987669074 | 100 |    0
(1 row)
```

2.8 Identify Duplicate Records

Note that there are 189118 `people_id` values, one for each row. We can take this to mean that there are no duplicate entries in the `people` dataset. The same is true with actions with 2695978 unique `act_id` values.

2.9 Search for NULL Data

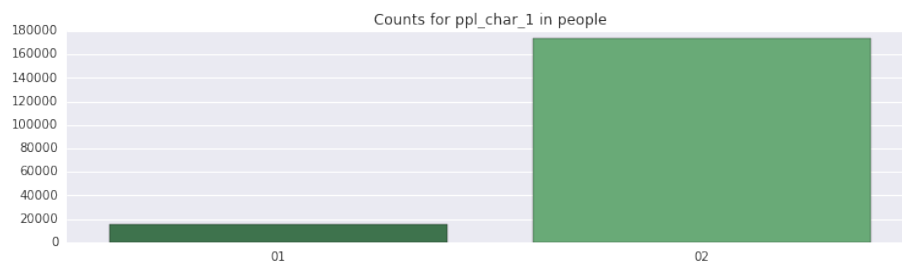
There is null data in these datasets, in two locations. There are null values in the boolean variables attached to the **action** table. We will be handling this data, however, when we process the data for handoff to the neural network. Additionally, there are null values in the **act_outcome** column, but this is functional as a null value in this field signifies a **test** action as opposed to a **train** action.

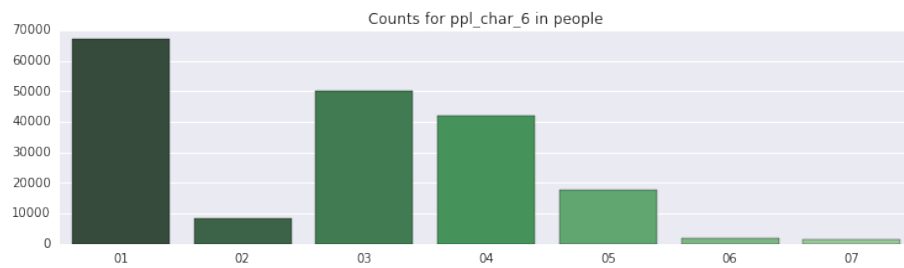
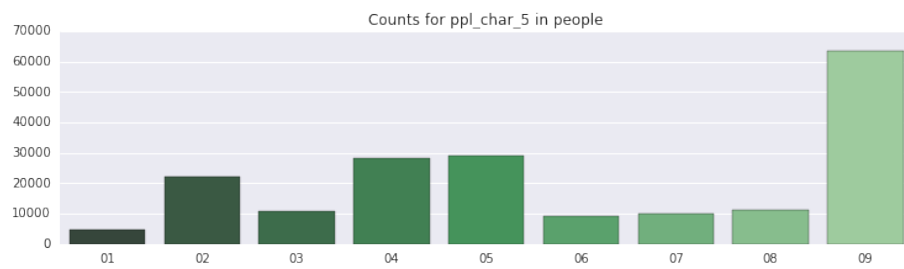
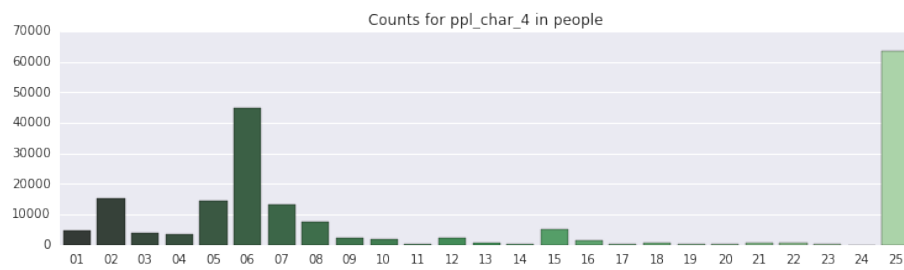
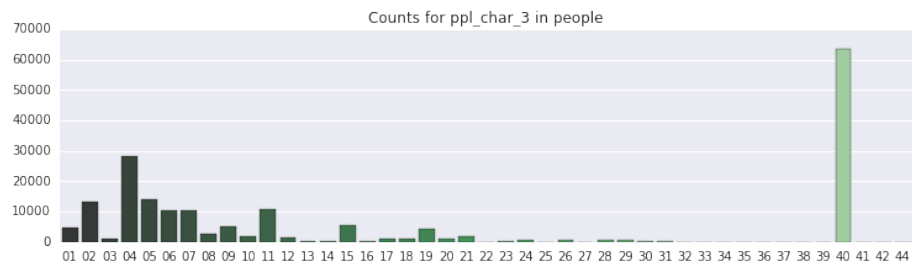
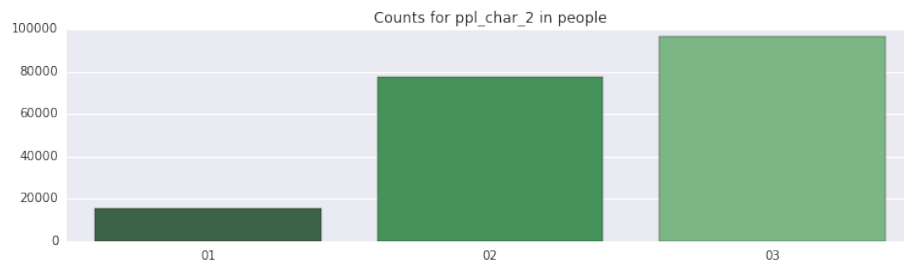
2.10 Exploratory Visualiztion

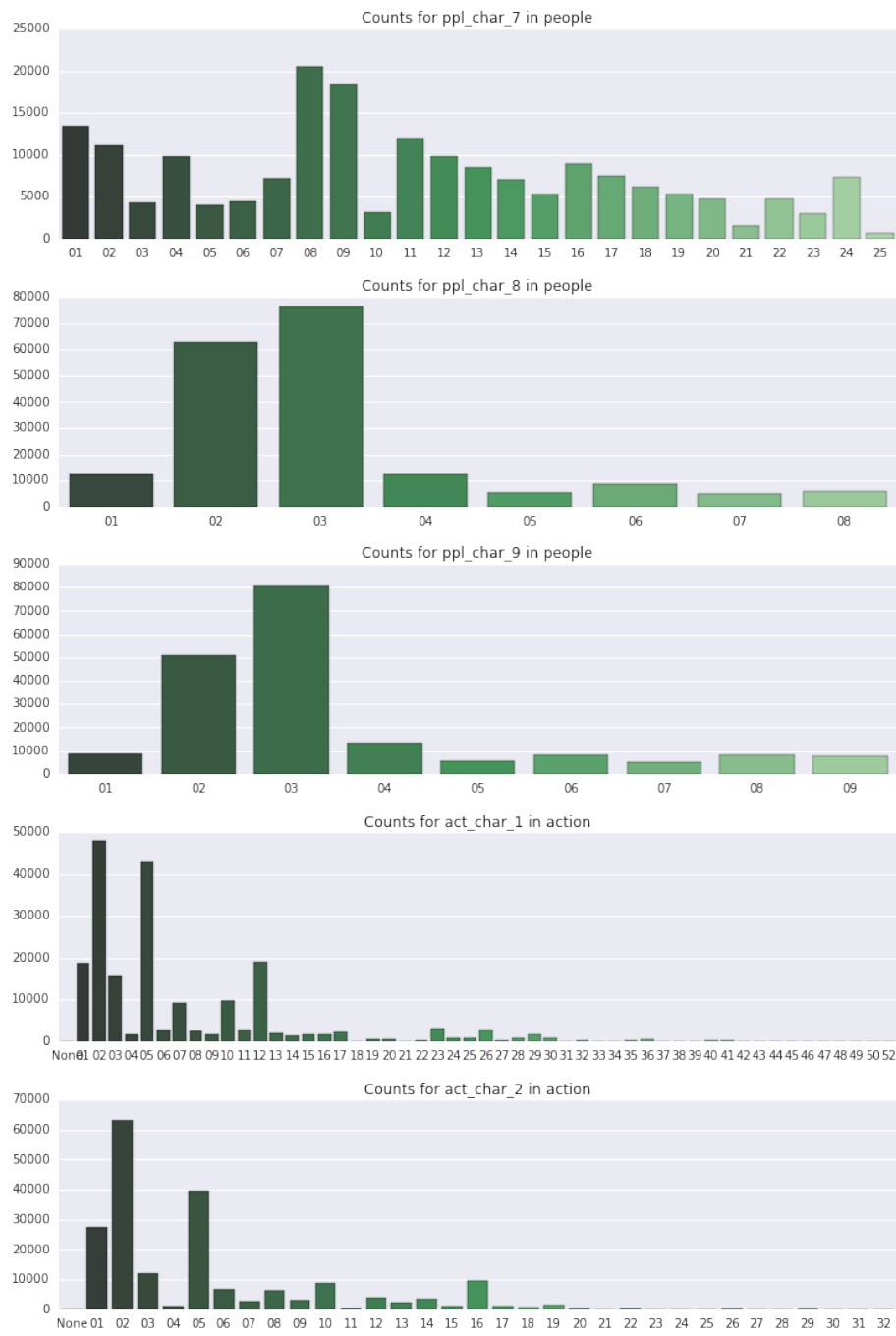
Please refer to notebook [2.10 Preliminary Data Analysis - Create Histograms of Data](#).

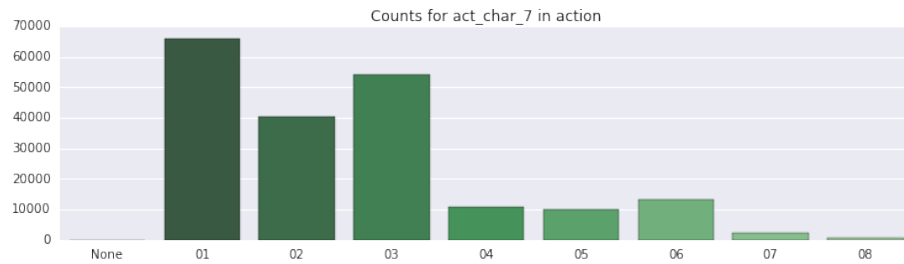
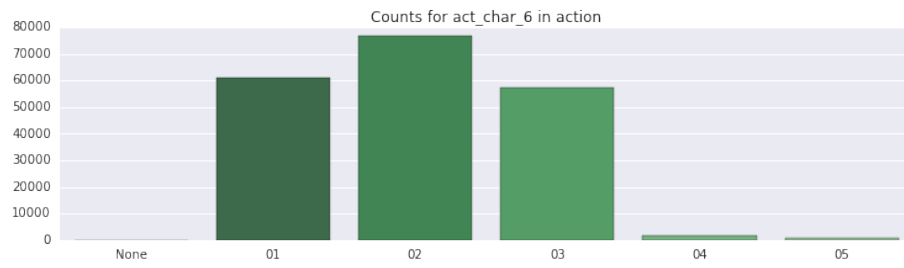
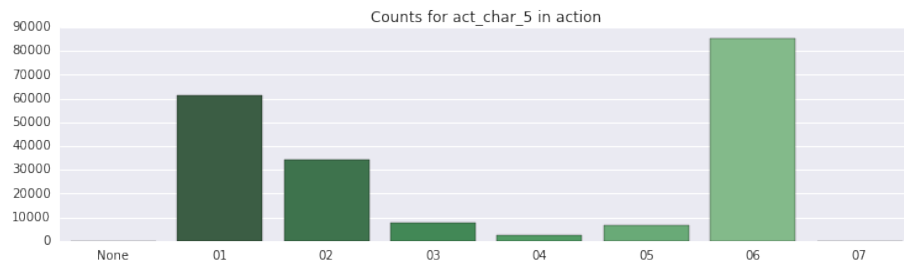
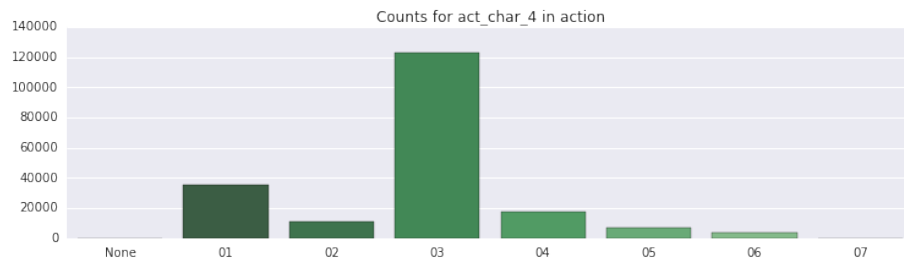
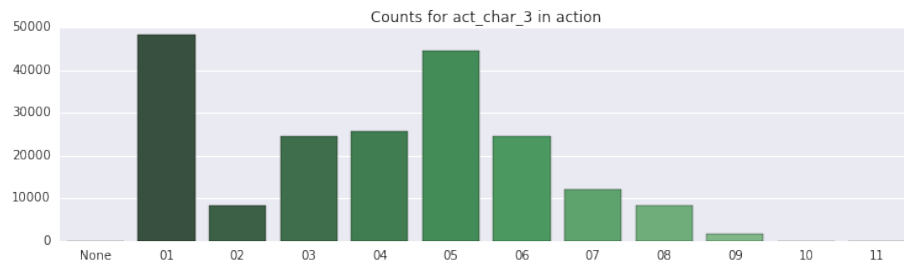
Finally, we use the Python library [seaborn](#) to create plots of our data as histograms. We import a method `bar_plot` to present a histogram for each categorical parameter.

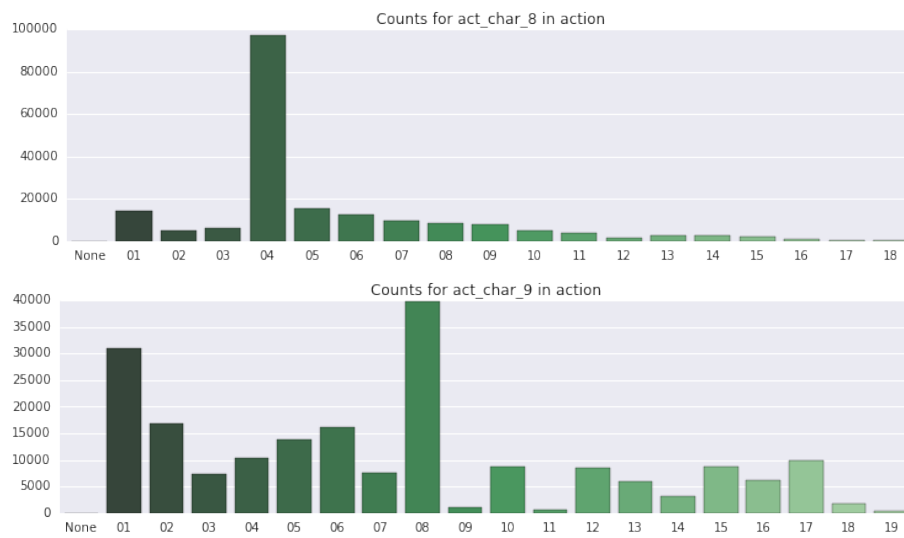
```
>>> from os import chdir; chdir('../')
>>> from lib.helpers.plot_helper import bar_plot
>>> bar_plot('ppl_char_1', 'people')
>>> bar_plot('ppl_char_2', 'people')
>>> bar_plot('ppl_char_3', 'people')
>>> bar_plot('ppl_char_4', 'people')
>>> bar_plot('ppl_char_5', 'people')
>>> bar_plot('ppl_char_6', 'people')
>>> bar_plot('ppl_char_7', 'people')
>>> bar_plot('ppl_char_8', 'people')
>>> bar_plot('ppl_char_9', 'people')
>>> bar_plot('act_char_1', 'action')
>>> bar_plot('act_char_2', 'action')
>>> bar_plot('act_char_3', 'action')
>>> bar_plot('act_char_4', 'action')
>>> bar_plot('act_char_5', 'action')
>>> bar_plot('act_char_6', 'action')
>>> bar_plot('act_char_7', 'action')
>>> bar_plot('act_char_8', 'action')
>>> bar_plot('act_char_9', 'action')
```











Chapter 3

Algorithms and Techniques

3.1 One-Hot Encoding

Please refer to notebook [3.01 Algorithms and Techniques - One-Hot Encoding Example](#).

We will use the One-Hot Encoding algorithm to convert our categorical data to numerical data. It may be tempting to merely convert our categories to numbers i.e. `type 01` \rightarrow 1, `type 02` \rightarrow 2, however, such an encoding of data implies a linear relationship between our categories, where there may be none.

In one-hot encoding, a separate bit of state is used for each state. It is called one-hot because only one bit is “hot” or TRUE at any time. (Harris, David, and Sarah Harris. Digital design and computer architecture. Elsevier, 2012.)

This algorithm is also referred to as 1-of-K encoding. An example will be helpful in illustrating the concept.

3.1.1 One-Hot Encoding Example

```
>>> import numpy as np
>>> from os import chdir; chdir('../')
>>> from lib.helpers.database_helper import connect_to_postgres
>>> conn, cur = connect_to_postgres()
>>>
>>> cur.execute("SELECT ppl_char_1,ppl_char_2 FROM people LIMIT 10")
>>> this_row = cur.fetchone()
>>> one_hot = []
>>> while this_row:
>>>     one_hot.append([
>>>         this_row[0] == 'type 1',
>>>         this_row[0] == 'type 2',
>>>         this_row[1] == 'type 1',
>>>         this_row[1] == 'type 2',
>>>         this_row[1] == 'type 3',
>>>     ])
>>>     this_row = cur.fetchone()
>>> print(np.array(one_hot, dtype=int))
[[0 1 0 1 0]
 [0 1 0 0 1]
 [0 1 0 0 1]
 [0 1 0 0 1]
 [0 1 0 0 1]
 [0 1 0 0 1]
 [0 1 0 0 1]
 [0 1 0 1 0]
 [0 1 0 0 1]
 [0 1 0 0 1]
 [0 1 0 0 1]]
```

Here, we select two columns from our database. For each available type for each column, we do a Boolean check and then cast this check to an integer. The result is that for a given group of columns corresponding to a single column in our original database, there will be a single 1 and the remainder will be 0. We use one-hot coding because the categorical and boolean nature of the vast majority of our data lends itself to this technique.

3.2 Linear Classification via Neural Network

Linear classification will be the core algorithm upon which we will build our neural network classifier. We borrow heavily for this approach from Andrej Karpathy's [notes](#) for his Convolutional Neural Networks course:

The approach will have two major components: a **score function** that maps the raw data to class scores, and a **loss function** that quantifies the agreement between the predicted scores and the ground truth labels.

3.2.1 Score Function

We will develop a score function that maps input vectors to class scores

$$f : \mathbb{R}^D \mapsto \mathbb{R}^2$$

where D is the dimension of our one-hot encoded vectors and 2 represents the 2 classes of our binary classifier. Then,

$$f(x_i, W, b) = Wx_i + b = y$$

where x_i is a particular input vector, W is a matrix of weights (dimension $2 \times n$), b is a bias vector, and y is a score vector with a score for each class.

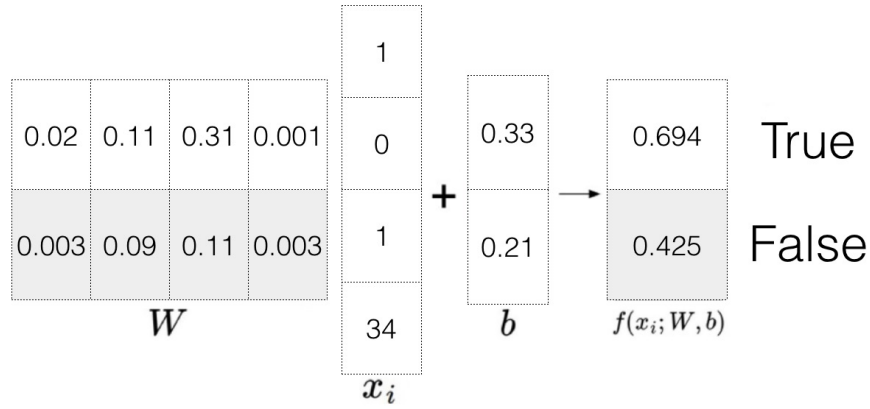


Figure 3.1: A Linear Classifier

3.2.2 Loss Function

Note that of the inputs to our score function we do not have control over the x_i s. Instead, we must change W and b to match a set of given y s. To do this we will define a loss function that measures our performance. We will use one of

the most common loss functions the multiclass support vector machine. Here the loss for a given vector is

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

Here, s is the vector result of our score function and y_i is the correct class. Our loss function computes a scalar value by comparing each incorrect class score to the correct class score. We expect the score of the correct class to be at least Δ larger than the score of each incorrect class.

3.2.3 Regularization Penalty

It is possible that more than one set of weights could provide an optimal response to our loss function. In order to prioritize the smallest possible weights we will add a regularization penalty to our loss function. Again we will go with a common technique and use the L2 norm.

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Additionally, including a regularization penalty has the added benefit of helping to prevent overfitting.

3.2.4 Final Loss Function

$$L = \frac{1}{N} \sum_i L_i + \lambda R(W)$$

Here, λ is a hyper parameter to be fit by cross-validation and N is a batch size.

3.3 Optimization

Possible methods:

3.3.1 Generate a Random Weights Matrix

- we initialize a weights matrix, W

3.3.2 Randomly guessing

- we initialize a weights matrix, W_{cur}
- for each vector (or batch of vectors) passed to the learner, we generate a new weights matrix, W_i
- if the new weights, W_i is better in score than W_{cur} , we assign it to W_{cur}

$$W_{cur} \rightarrow W_i$$

- repeat for all of our test vectors

3.3.3 Random Local Search

- we initialize a weights matrix, W
- for each batch of vectors passed, we generate a random matrix, ΔW , of the same dimension as W and scaled by some factor, ν
- we measure the loss against the sum $W + \nu\Delta W$.
- If $W + \nu\Delta W$ has a better score than W , we assign it to W

$$W + \nu\Delta W \rightarrow W$$

- repeat for all of our test vectors

3.3.4 Gradient Descent

- compute the best direction along which we should change our weight matrix that is mathematically guaranteed to be the direction of the steepest descent
- the gradient is a vector of derivatives for each dimension in the input space
- calculate the gradient and use this calculation to update the weight matrix

$$W_{new} = W - \nabla L$$

3.4 Benchmark

Of note is that, while the outcome is clearly defined by the contest, for the purposes of this project, we will be using a portion of the training set as our benchmark. Of note is that we modified our performance metric during the refinement phase.

As previously noted, we will look at three different properties of our test set in order to measure the success of our learner:

1. a confusion matrix
2. accuracy
3. F1 Score

It is difficult to establish an appropriate benchmark given the nature of the codebase for this project. The vast majority of the code here has been written using pure numpy. As such, we can not hope to compete with the latest libraries such as XGBoost or Keras. A random guessing learner would be successful 50% of the time. We think it a reasonable goal that our methods written by hand achieve an 80% accuracy.

3.4.1 Confusion Matrix

A confusion matrix is a table that allows the visualization of the algorithm performance. Each row represents the actual classes of our target variable: 1 or 0. Each column represents the predicted classes of our target variable: 1 or 0. We will then measure the true and false positives as well as the true and false negatives.

3.4.2 Accuracy

Accuracy will be calculated in the following manner:

$$\text{Accuracy} = \frac{\text{True Postives} + \text{True Negative}}{\text{Postives} + \text{Negatives}}$$

We will be trying to maximize this value.

3.4.3 F1 Score

F1 Score will be calculated as

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

where

$$\text{precision} = \frac{\text{True Positives}}{\text{True Positive} + \text{False Positives}}$$

and

$$\text{recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

We will be trying to maximize this value.

Chapter 4

Free-Form Visualization

4.1 Visualizing the Loss Function

Please refer to notebook [4.01 Exploratory Visualization - Visualizing the Loss Function](#).

A relevant visualization to this task is that of the loss function. For this visualization, we again turn to Andrej Karpathy's [notes](#).

While we will have difficulty visualizing the loss function over the complete weight space, we can visualize it over a smaller space to begin to understand our approach.

```
>>> import numpy as np
>>> from os import chdir; chdir('../')
>>> from lib.helpers.viz_helper import loss_function_i, \
                                         loss_function_in_a_direction, \
                                         render_all_plots_1d, \
                                         render_all_plots_2d
```

For the purposes of this visualization, let us consider a small random weight matrix $(2, p)$ for a binary classifier, i.e., one weight vector for each classifier.

```
>>> W = np.random.rand(2,7)
```

We then generate a random input vector x (with 6 parameters, and then a trailing bias) and a vector of outputs.

```
>>> x = np.random.randint(2, size=7)
>>> x[6] = 1
```

Finally, we randomly select a correct outcome for a binary classifier.

```
>>> correct_class = np.random.randint(2)
```

We vary the loss function for a single input with different weights for a single parameter, `param`, then plot this function along various values of `variable_weight` for all of our `params` values.

```
>>> render_all_plots_1d(correct_class,x,W)
```

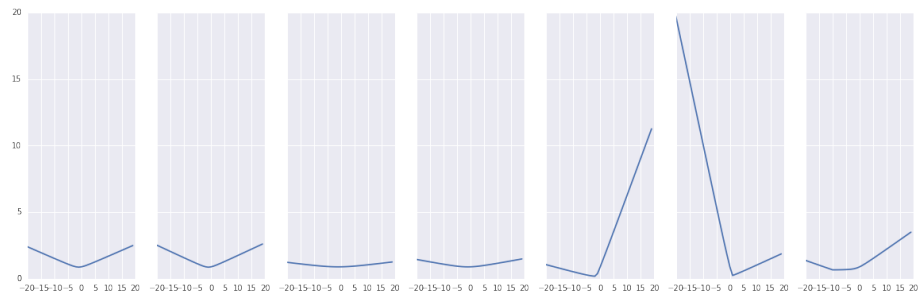


Figure 4.1: Visualizing Loss Function change along one Parameter

It is of note that every parameter is convex and can be minimized.

We can also do the same for a comparison of two varied parameters. Again, note that each of these plots is convex.

```
>>> render_all_plots_2d(correct_class,x,W)
```

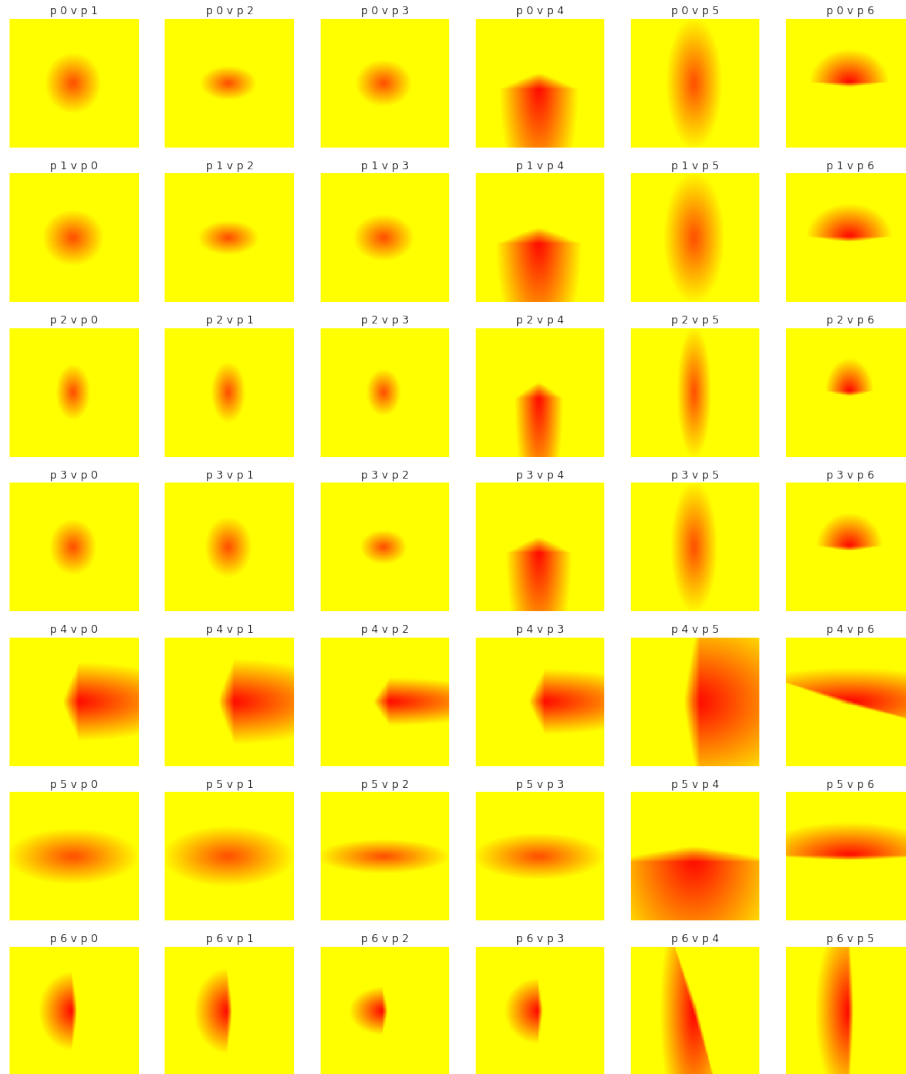


Figure 4.2: Visualizing Loss Function change against Two Parameters

Chapter 5

Data Preprocessing

5.1 CSV Manipulation

The dataset was a set provided by Kaggle. As such, it was already well structured and clean. Still, in order to facilitate processing, some work had to be done on the csv data itself.

5.1.1 `act_train.csv`

An additional column had to be added to the csv in order to ultimately provide a null space in which to insert our `act_one_hot_encoded` binary value. This was done via the `sed` command line tool by adding a comma to each line.

```
$ sed -e 's/$/,/' -i act_train.csv > new_act_train.csv
```

5.1.2 `act_test.csv`

For the test data set, we needed to add two columns, one for the null outcome (test and train are stored in the same table and distinguished by having a true, false or null value) and the same null space in which to insert the `act_one_hot_encoded` binary value.

```
$ sed -e 's/$/,/' -i act_test.csv > new_act_test.csv
```

5.1.3 All Sets

Additionally, we wanted to convert all attributes to double digit attributes i.e. `char 1` \rightarrow `char 01`.

```
$ sed -e 's/,char (\d),/,char 0\1,/' -i act_train.csv > new_act_train.csv
```

In this section, all of your preprocessing steps will need to be clearly documented, if any were necessary. From the previous section, any of the abnormalities or characteristics that you identified about the dataset will be addressed and corrected here. Questions to ask yourself when writing this section:

5.2 One-Hot Encoding

We will be storing our one-hot encoded numpy arrays as binary data in the action table column `act_one_hot_encoded`.

Chapter 6

Implementation

6.1 Steps to Implementation

1. Seed a PostgreSQL database with the three csv files.
2. One-Hot Encode the data and store the one-hot encoded vector as an array in the `action` table
3. Train and Assess a Series of Learners

6.2 Seed a PostgreSQL database with the three csv files.

This step is done at instantiation of the system. Refer to [Seeding the Database](#).

6.3 One-Hot Encode the data and store the one-hot encoded vector as an array in the `action` table

Please refer to notebook [6.03 Implementation - Write One-Hot to Action Table](#).

```
>>> from os import chdir; chdir('../')
>>> from lib.app import Q
>>> from lib.helpers.database_helper import connect_to_postgres
>>> from lib.helpers.database_helper import pull_actions_and_one_hot_encode
```

```
>>> for i in range(1000):
    Q.enqueue(pull_actions_and_one_hot_encode, 1000, i*1000)

>>> conn, cur = connect_to_postgres()
>>> cur.execute("SELECT count(*) FROM action where act_one_hot_encoding is not null;")
>>> cur.fetchone()
(372666,)
```

We have written a library to handle the one-hot encoding of the data. `pull_actions_and_one_hot_encode` does a join on the action and people tables, converts the tables and categories to one-hot encoded data, converts this to a binary `numpy` vector, and writes this binary to the action table, for actions from the action table that do not yet have one-hot encoded vectors.

Note that we are also using our delayed job system to do the conversion. Once jobs have been enqueued, the status of enqueued jobs can be tracked [here](#).

6.4 Prediction on Random Weights Matrix With No Training

Please refer to notebook [6.04 Implementation - Prediction on Random Weights Matrix With No Training](#).

We first establish a baseline competency by examining performance of a totally untrained learner.

```
>>> from os import chdir; chdir('../')
>>> from random import shuffle, seed
>>> from lib.helpers.database_helper import pull_actions, pull_and_shape_batch
>>> from lib.nn.metrics import measure_accuracy, measure_f1_score, correlation_matrix
>>> from lib.nn.functions import random_matrix, predict
```

6.4.0.1 Pull Training and Test Rows

We `seed` the shuffling mechanism for deterministic results. We then pull a set of 90000 `action_ids` from the action table. We shuffle these ids and designate the first 75000 as our training set and the last 15000 as our test set. To reiterate, we are not using the Kaggle competitions test set as we do not have the actual outcomes for that set and we can thus not measure the accuracy of our learner.

6.4.0.2 Train Learner and Assess Learner Accuracy

We are not actually training a learner here. We are merely generating a random matrix of the appropriate size. We then check the accuracy of this random matrix against our test set. This is done four times via the `%%timeit` ipython magic function.

```
>>> seed(42)
>>> action_set = pull_actions(limit=90000,action_type='training')
>>> shuffle(action_set)
>>> training_set = action_set[:75000]
>>> test_set = action_set[75000:]

%%timeit
>>> # initialize a random_weights matrix
>>> random_weights = random_matrix(2, 7326)
>>> features, outcomes = pull_and_shape_batch(action_ids=test_set)
>>> predicted = predict(random_weights, features)
>>> correlation_matrix(predicted, outcomes)
>>> print(measure_accuracy(predicted, outcomes))
>>> print(measure_f1_score(predicted, outcomes))
```

	act true	act false	totals
pred true	4971	2134	7105
pred false	2118	5777	7895
totals	7089	7911	15000

```
accuracy: 0.716533333333
f1_score: 0.7004368042835001
```

So this purely random matrix performs quite well, achieving an accuracy of 71.7% and an F1-score of 0.700.

We run the same process a second time and third time.

6.4.1 Second Pass with a Random Matrix

	act true	act false	totals
--	-------------	--------------	--------

pred true	39	1624	1663
pred false	7050	6287	13337
totals	7089	7911	15000

accuracy: 0.421733333333
f1_score: 0.008912248628884827

6.4.2 Third Pass with a Random Matrix

	act true	act false	totals
pred true	0	2	2
pred false	7089	7909	14998
totals	7089	7911	15000

accuracy: 0.527266666667

The third pass actually returned a `ZeroDivisionError` for the F1 Score.

6.5 Results from Training on a Random Matrix

Training on a random matrix is highly unstable and not a suitable method for learning.

Chapter 7

Refinement

Slightly better than guessing is a pretty poor performance. In order to improve upon our performance, we will try a series of improvements upon our learner in order to obtain better performance.

7.1 Learning via Random Search

Please refer to notebook [7.01 Refinement - Learning via Random Search](#).

As a first attempt at improvement, we will work in batches through our training set. For each batch we will:

1. generate a random weights matrix
2. evaluate the random weight matrix against the loss function
3. if the loss function is lower than the previous lowest loss function
4. store the loss function as the `best_loss` and the weights matrix as `weights_matrix`

It is of note that we will be doing the training via distributed processing. As such, we can not store the `best_loss` and `weights_matrix` in memory. Instead, we store the values in Redis. We have written a few methods to handle the storage and retrieval of these values.

- `read_best_loss`
- `read_weights_matrix`
- `write_best_loss`
- `write_weights_matrix`

7.1.1 Establish State of Training Session

```
>>> training_set, test_set = pull_training_and_test_sets(limit=700000,split=.99)
>>> features, outcomes = pull_and_shape_batch(action_ids=test_set)
>>> initialize_training_session()
>>> batch_size = 50
>>> for i in range(int(len(training_set)/batch_size)):
>>>     Q.enqueue(train_via_random_search,
>>>                 action_ids=training_set[i*batch_size:(i+1)*batch_size],
>>>                 gamma=0.001)
>>> counts, f_1_scores, \
>>>     accuracies, \
>>>     loss_values = prepare_plot_vectors(features,
>>>                                         outcomes,
>>>                                         length=int(len(training_set)/batch_size))
```

7.1.2 Assess Results

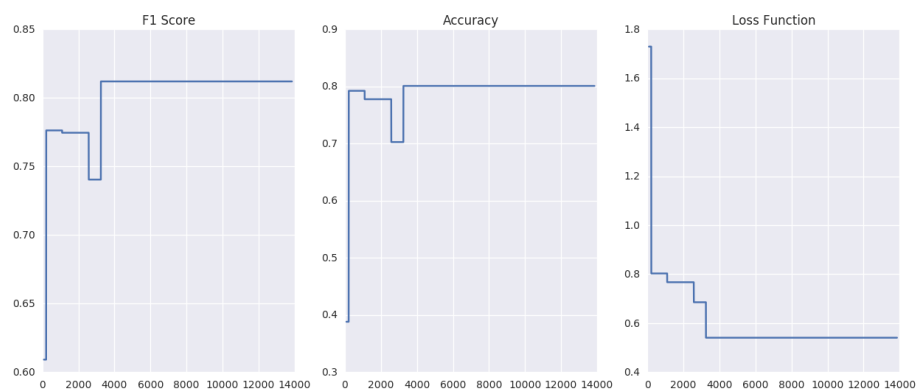


Figure 7.1: Training via Random Search

7.2 Learning via Random Local Search

Please refer to notebook [7.02 Refinement - Learning via Random Local Search.ipynb](#).

Next, we attempt to improve our initial random matrix via microchanges in the local vicinity. Again, we will work in batches through our training set. For each batch we will:

1. generate a random weights matrix delta
2. add the delta to our current weights matrix
3. evaluate the temporary weight matrix against the loss function
4. if the loss function is lower than the previous lowest loss function
5. store the loss function as the `best_loss` and the temporary weights matrix as `weights_matrix`

7.2.1 Establish State of Training Session

```
>>> training_set, test_set = pull_training_and_test_sets()
>>> initialize_training_session()
>>> for i in range(int(len(training_set)/100)):
        Q.enqueue(train_via_random_local_search,
                    action_ids=training_set[i*100:(i+1)*100],
                    gamma=0.001)
>>> training_counts, loss_values = prepare_plot_of_loss_function()
```

7.2.2 Assess Results

```
>>> weights_matrix = get_weights_matrix()
>>> features, outcomes = pull_and_shape_batch(action_ids=test_set[:100])
>>> measure_accuracy(weights_matrix, features, outcomes)
0.64000000000000001
```

Here, we actually have a lower accuracy result. This largely speaks to the instability of this method.

If we look at the plot of the loss function over the training, we can see that there was significantly greater improvement over the course of the training using this method.

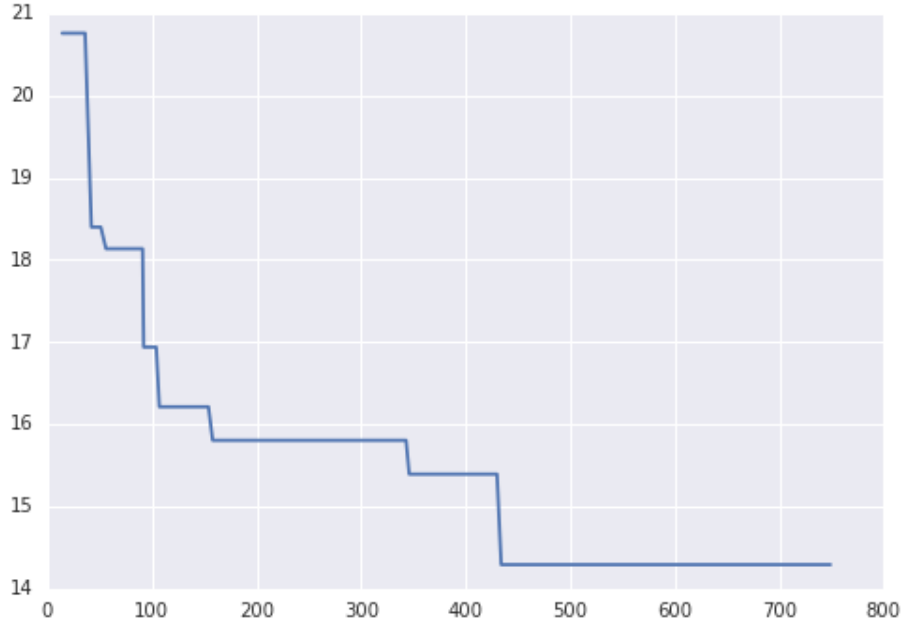


Figure 7.2: Loss Function over Random Local Search Training

7.3 Learning via Gradient Descent

Please refer to notebook [7.03 Refinement - Learning via Gradient Descent](#).

As one might imagine there is actually a better for local optimization than a random step. We will improve upon our prediction by taking a step in the *optimal* direction at each learning phase. The loss function is the function that we are attempting to minimize. The gradient of this loss function will tell use the direction in which the loss function is most rapidly decreasing.

Consider our loss function in one-dimension (without regularization):

$$L_i = \sum_{j \neq y+i} \max(0, s_j - s_{y_i} + \Delta)$$

Where s_i is the score for our correct class and s_j is the score for incorrect class(es).

We obtain the gradient by differentiating with respect to the weights to obtain, for the row of the weight matrix corresponding to the correct class:

$$\nabla_{w_{y_i}} L_i = - \left(\sum_{j \neq y_i} \mathbb{K}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i$$

and for the row(s) of the weight matrix corresponding to the incorrect class(es):

$$\nabla_{w_j} L_i = \mathbb{K}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

Here \mathbb{K} is the indicator function that is one if the condition inside is true or zero otherwise.

7.3.1 Establish State of Training Session

```
>>> n = int(len(training_set)/10)
>>> initialize_training_session()
>>> for i in range(n):
    Q.enqueue(train_via_gradient_descent,
               action_ids=training_set[i*10:(i+1)*10],
               gamma=0.001)
>>> training_counts, loss_values = prepare_plot_of_loss_function(length=n)
```

7.3.2 Assess Results

```
>>> weights_matrix = get_weights_matrix()
>>> features, outcomes = pull_and_shape_batch(action_ids=test_set)
>>> measure_accuracy(weights_matrix, features, outcomes)
0.42180000000000001
```

This is the first training method not based on random chance. We note:

- it has performed very poorly
- we see the exponential growth of our loss function

At this point, we did some deep consideration of our approach. The exponential growth of the loss function is a problem of implementation. We are not minimizing with respect to a regularized loss function. This is not necessarily the cause of our poor performance. Most likely, the cause of our poor performance is the curse of dimensionality.

`act_char_10` has 6969 unique labels. In order to properly one-hot encode this column, we require as many binary features. This one column multiplies the size of our features space by seven. Given that complexity grows exponentially, we are pinning the poor performance of our learner on this column and discard it for our next pass.

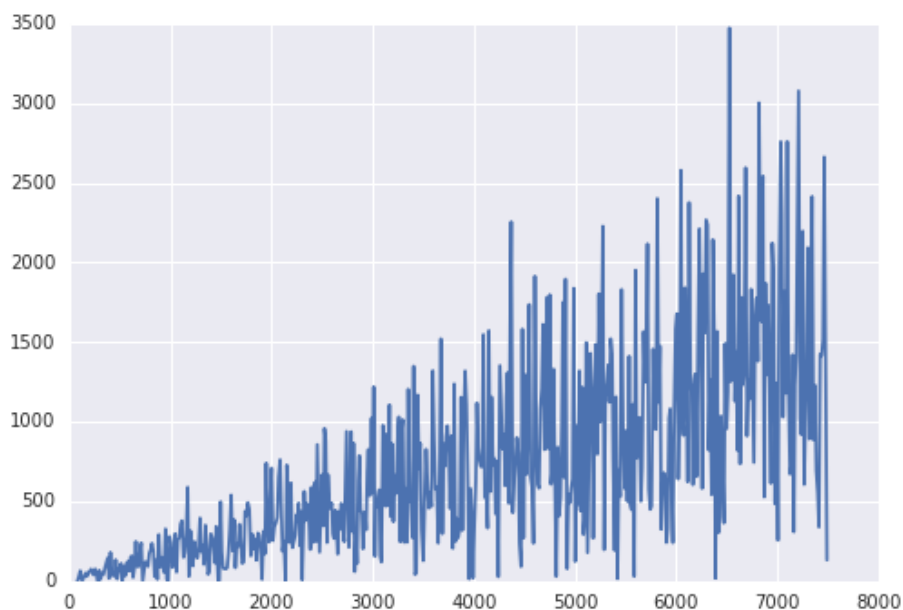


Figure 7.3: Loss Function over Gradient Search Training

7.4 Recover from Curse of Dimensionality

Please refer to notebook [7.04 Refinement - Recover from Curse of Dimensionality](#).

In order to drop `act_char_10`, we must change the way that we are performing our one-hot encoding. We create a new table of columns called `one_hot_indices_min.py`.

7.5 Rewrite One-Hot to Action Table

Please refer to notebook [7.05 Refinement - Rewrite One-Hot to Action Table](#).

7.6 Learning through a multi-layer Neural Network

At this point, we hit a wall. The complexity of managing the state of our learner via a Redis learner and the processing of learning jobs via a distributed network

of python servers proved to be insurmountable at this time. The implementation of a multi-layer neural network requires considerable investment in our system architecture that is not feasible at this time.

Please refer to notebook [7.03 Refinement - Learning via Gradient Descent](#).

Chapter 8

Results

8.1 Final Model

8.2 Benchmark Comparison

Chapter 9

Conclusion

9.1 Summary of Project

This project allowed me to develop many tools that I will use again. Perhaps the greatest success of this project is the portability of data provided by the `docker-compose` tool in conjunction with the Postgres native docker image. In my past work, the data itself has been the least portable aspect of the project, and using this tool I have been able to run this project on no less than four different systems. This has allowed my great flexibility in trying different hardware configurations. This will prove instrumental in developing pipelines for solving additional kaggle tasks in the future.

Additionally, the `rq` worker system has been a tremendous success. When I first conceived of this idea, using delayed job processing as a means of accessing multiple cores on a single cpu or in the interest of doing work across a cluster, I was not certain that it would work. Not only did it work, it has given me many ideas upon which I can iterate going forward.

Building the various learners by hand gave me great insight into the math underlying a perceptron learner and a network of perceptron learners. While I doubt I will ever use this code again, I learned a great deal in the implementation and am better algorithmist as well as programmer for having done this work.

In building our learner, we iterated through several phases: pure randomness, local randomness, stochastic gradient descent and neural networks. It was very illuminating to see the results take shape at each new phase.

In terms of what didn't work, the worker system is limited in what it can do. It can not distribute the multiplication of a single matrix across many cores. In this sense, I haven't really dealt with a core multi-threading problem. I think in the future, I will look into the Blaze ecosystem. This is a library that is being

developed by Continuum Analytics. I think here I shall stand on the shoulders of giants.

In terms of the learners, we did not see amazing results. A lot of this has to do with just biting off an enormous data set at the same time as I was trying to write my own libraries from scratch. I think that had I chosen to use an open-source library designed to work on large data sets, such as TensorFlow, we would have seen much greater results.

Lastly, I was never able to get the cluster to work correctly. I think that the Docker Swarm technology I have been attempting to leverage is not quite ready and is too much in a beta state. I literally say the technology change while I was working on the project and my early code would no longer run. I think I would like to implement a more stable solution to clustering such as the Kubernetes library.

9.2 Steps for Improvement

I think we can look at three main ways to improve this project. All three have to do with looking to outside libraries.

1. I do not need to reinvent high performance computing, especially given the fact that I have no viable solution for distributed matrix computations. The solution to this is use Dask, part of the Blaze ecosystem.
2. Docker Swarm is not ready for prime time. Use Kubernetes for clustering. It is robust and fully supported.
3. Writing code from scratch is a worthwhile exercise and an excellent cap to program, but next time use TensorFlow.

9.3 Reflection

It is important to note that nearly every line of code in this project was written by me. In this sense, there has been great value in attempting to implement some of the most challenging machine learning concepts from scratch. That said, it is clearly not the best solution to this task. Tensorflow or a similar library should have been brought to bear on this task. It is of note that tensorflow has been developed by a team of more than 20 people. While I have learned a lot in writing this code base, especially in terms of writing clean Python, perhaps the most important thing I have learned is that it is okay to stand on the shoulders of giants. I don't need to do everything from scratch.

This project attempted to tackle quite a bit. I was attempting to engineer a robust system that could handle a large data set. In the end, my decision to hand

code everything as opposed to using libraries gave access to high performance that I might not have had otherwise, but in spending time on the engineering, I never quite got to a point where I was iterating on the algorithms themselves.