

Table of Contents

Get Data.....	2
Clean and Prepare	4
<i>First-Pass Verification</i>	4
<i>Data Types</i>	5
<i>Nulls</i>	7
Feature Inspection	9
<i>Continuous Features</i>	9
<i>Categorical Features</i>	12
<i>Date Features</i>	12
Feature Scaling and Other Preprocessing.....	13
<i>Feature Scaling and Continuous Features</i>	13
<i>Categorical or Boolean Features</i>	14
<i>Using Optimized Data Structures</i>	14
Feature Selection	16
<i>Correlation</i>	16
<i>Lasso Feature Selection</i>	17
<i>Random Forest Feature Importance</i>	19
<i>Recursive Features Elimination</i>	19
Model Horse Race	22
Output Interpretation	Error! Bookmark not defined.

Get Data

We start by setting the path of the working directory relative to home directory so when you go to other machines, assuming you have the same file structure, you don't have to modify your code. With the **os library**, the HOME variable is after /User/<your_user_name>.

```
# Get Data and Format File
# set current working directory
WkDir = os.path.join(os.environ['HOME'], 'personlich/cronin_guidebook/')
os.chdir(WkDir)
os.getcwd()
```

Use the **json library** to read strings in as a json object. The json library primarily converts json objects to strings and vice-versa.

```
# get file
JsonFile = open('rideshare_user_history.json', 'r').read() # reads text of file as string
JsonLoad = json.loads(JsonFile) # converts string to parsed json

# read file head
JsonLoad[:5]

# convert list of dictionaries to pd.DataFrame
Df = pd.DataFrame(JsonLoad)
DfBackup = Df.copy()
```

Because json is a flexible data structure some entries can contain keys you didn't expect. Something I do when dealing with a new json field is flatten it by identifying every key listed in every single json object, and print it out with counts for # of occurrences. We start by **identifying all the unique keys** in the json blob then initialize a dataframe with each key for a column.

```
# Flatten Dictionaries in Case Different Number of Keys
# in case file dictionaries have different sets of keys
JsonChklst = []
for blob in JsonLoad:
    for key in blob.keys():
        JsonChklst.append(key)
UnqKeys = set(JsonChklst)

# initialize df with UnqKeys for columns
Df = pd.DataFrame(columns=UnqKeys)
```

Even though the json blob file only contains 50k rows **df.append()** is an expensive process because every time a new row is appended, the index of the pd.DataFrame is deleted and rewritten. In the future instead of this I would create do this a list of lists (for example:

[[row1], [row2], ..., [rowN]] to later convert to a DataFrame. For this reason I test with 3 rows then build out to the full 50k:

```
# populate 3 rows as test
for blob in JsonLoad[:2]:
    LolNewVals = [blob.values()]
    ListNewKeys = blob.keys()
    DfNewRow = pd.DataFrame(data=LolNewVals, columns=ListNewKeys)
    Df = Df.append(DfNewRow, ignore_index=True)

# # populate all rows
for blob in JsonLoad:
    LolNewVals = [blob.values()]
    ListNewKeys = blob.keys()
    DfNewRow = pd.DataFrame(data=LolNewVals, columns=ListNewKeys)
    Df = Df.append(DfNewRow, ignore_index=True)
```

Clean and Prepare

The goals of initial cleaning and preparation are two-fold: make sure the data you took in is what you expect it to be, and format data so it's ready for analysis.

First-Pass Verification

We start by looking at the dataframe at a very high level. Shape, head, datatypes. Use shape to make sure you got all the rows. Use head to make sure the values read in correctly. Often you'll find dates were misread or a field wasn't correctly read in, so populated with NaNs.

```
# Inspect Table  
# check size of frame: 12 cols, 21k rows  
Df.shape  
  
# check df head  
Df.head()  
  
# check data types  
Df.dtypes
```

Data Types

These are the standard steps for converting each type of variable to a model-friendly format:

	Transformation	Useful Functions
Date	String → Date → Numeric	pd.to_datetime()
Categorical	String → Category → Dummies	Df['field'].astype('category') new_field = Pd.get_dummies(['field']) Pd.concat([Df, new_field], axis=1)
Boolean	String → True/False	Np.where()
Ordinal	String → Category → Level Mapping	Df['field'].astype('category') Map = {label: level} Df['field'].map(Map)
Categorical	String → Category → Dummies	Df['field'].astype('category') new_field = Pd.get_dummies(['field']) Pd.concat([Df, new_field], axis=1)

In General:

- Models understand **ordinal values** that are monotonically increasing because they can be represented by numbers. For example you can encode t-shirt sizes, S/M/L as 1/2/3 because each subsequent size is *necessarily larger* than the last one. This is where the **.map()** function becomes handy.
- Models don't understand **categorical values** because there is no order to them. The workaround is to convert categories into many dummy variables. For example if we have the categories iPhone/Android/Windows for mobile phone Operating System types, you would convert the feature into three separate ones; one for each type of phone. The pd.get_dummies() function performs this task.
- **Dates** take special attention. As far as computers are concerned, dates are integers. For example unixtime (also called epoch time) is the number of seconds that have passed since some arbitrary date, 1970-01-01. Under every date is an integer representing some unit of time. Therefore when working with dates in models, you have to convert them into something interpretable. Day of week = categorical, month = categorical, year = ordinal (maybe), date = days from X event, and so forth.

In the case of this challenge exercise, we'll focus on dates first. Luckily you can take these in without much trouble because the original data is formatted nicely—YYYY-MM-DD. After creating our date fields, we can create our churn field and other related fields (e.g. day of week, month of year, quarter, etc.). The **datetime library** is the go-to for flexible date formatting and conversions. Dates are typically the most difficult datatype to deal with in any language and there are libraries to deal with every date conversion you'll come across, including truncation, epoch time conversion, and timezone conversion.

```
# dates
Df['last_trip_date'] = pd.to_datetime(Df['last_trip_date'])
Df['signup_date'] = pd.to_datetime(Df['signup_date'])

# create target variable (based on date field)
TimeCutoff = Df['last_trip_date'].max() - dt.timedelta(days=30)
Df['churn'] = np.where(Df['last_trip_date'] < TimeCutoff, 1, 0)

# convert dates to numeric
date_diff = (Df['signup_date'] - Df['signup_date']).min()
Df['days_from_first_signup'] = date_diff.astype('timedelta64[D]').astype('float')
```

Next we address categorical, boolean (special case of categorical), and continuous variables. You'll notice with `trips_in_first_30_days` all I did was cast the datatype to float. This is so down the road any transformations or functions will treat all features of the same datatype in exactly the same way.

```
# categorical
Df['city'] = Df['city'].astype('category')
Df['phone'] = Df['phone'].astype('category')

# boolean
Df['black_car_user'] = np.where(Df['black_car_user'] == True, 1, 0)

# continuous (percents)
Df['weekday_pct'] = Df['weekday_pct'].astype('float')/100
Df['surge_pct'] = Df['surge_pct'].astype('float')/100
Df['trips_in_first_30_days'] = Df['trips_in_first_30_days'].astype(np.float)
```

Nulls

Nulls require careful attention. Sometimes they're anomalies, sometimes they're missing for a good reason and it doesn't make sense to throw a row out just because it's missing one value. Once you have a better sense for why a value is missing, you can fill it in with a value that makes sense given your hypothesis for why it's missing. Here we start by looking at the number of % of rows per feature:

```
# inspect nulls
Df.isnull().sum() # total nulls
Df.isnull().sum()/len(Df) # percent nulls
```

avg_dist	0.00000
avg_rating_by_driver	0.00402
avg_rating_of_driver	0.16244
avg_surge	0.00000
black_car_user	0.00000
city	0.00000
last_trip_date	0.00000
phone	0.00792
signup_date	0.00000
surge_pct	0.00000
trips_in_first_30_days	0.00000
weekday_pct	0.00000
churn	0.00000
days_from_first_signup	0.00000

Because there are seemingly important features with a relatively large number of missing values (avg_rating_of_driver ~ 16.2% missing values) we should create two data sets, one with complete rows and one with all rows. The one with all rows we'll either fill in or drop cases as we better understand why there are nulls.

```
# split analysis with two dataframes, one with dropped values one with imputed
DfImp = Df.copy() # create Df for imputing values
DfCmp = Df.dropna(axis=0) # create Df for dropping rows
```

Start by printing summary stats on the complete cases. Develop a baseline/expectation for "normal behavior."

```
# look at df with complete entries
DfCmp.describe()
```

Let's start by analyzing avg_rating_of_driver because nulls exist in 16% of cases. We start by eyeballing summary stats to compare against the complete cases in DfCmp.

```
# look at cases where avg_rating_of_driver is null ==> 16% of cases so must attempt imputing
NullArod = DfImp['avg_rating_of_driver'].isnull()
DfImp[NullArod].describe() # looks like all riders had 0 or 1 rides in first 30
```

The key difference appears to be in trips_in_first_30_days. The other fields, while slightly different, have mean and standard deviation that fall well within range of each other—meaning there aren't any **statistically significant differences**. It's worth inspecting trips_in_first_30_days visually by their comparing their distributions.

```
# inspect difference in distributions for first 30 days
x_missing = DfImp[NullArod].trips_in_first_30_days
x_complete = DfCmp.trips_in_first_30_days

bins = np.linspace(0, 20, 100); nm=1; al=0.8; wid=0.5
plt.hist(x_missing, bins, normed=nm, alpha=al, width=wid) # hist of first 30 day rides for missing rows
plt.hist(x_complete, bins, normed=nm, alpha=al, width=wid) # hist of first 30 day rides for complete rows
plt.show()
```



It appears these users that didn't provide reviews were fast churners. They almost unanimously only recorded 0 or 1 rides in their first month. It's possible those with 0 rides later took 1 ride but churned after and didn't provide a rating. In order to keep their data, **impute the missing values to the mean**. Doing this least affects model fits because you're just adding **central tendency** to the field. Later we'll see if our model fits substantially differently on complete cases, to see if this was the right decision.

The phone and avg_rating_by_driver summary stats also look very similar to the complete cases stats, so we impute those to average or "Other." The decision on what to do here has much less impact on the outcome of our analysis since they represent so few cases.

We do one final check to make sure we've addressed all our nulls:

```
# did we get rid of all the nulls?
DfImp.isnull().sum()
```

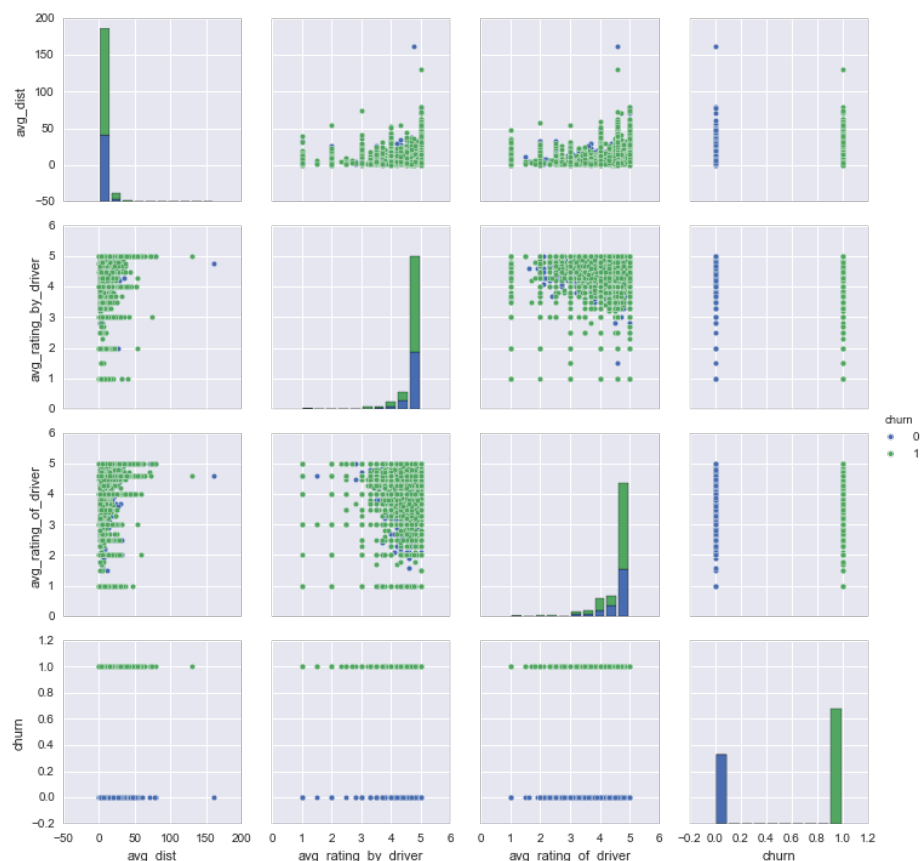

Feature Inspection

After getting data into usable format and figuring out what to do with missing values, we can start doing some purposeful exploratory analysis. The primary questions here are, “how do features relate to each other?” “Are there skews or other anomalies we need to address?”

Continuous Features

Starting with continuous variables, we identify the continuous fields then do pair plots to look at how our features relate to each other, and how well they separate your target variable, in this case, churn.

```
# look at all continuous features
ColsCont = [
    'avg_dist',
    'avg_rating_by_driver',
    'avg_rating_of_driver',
    'avg_surge',
    'surge_pct',
    'trips_in_first_30_days',
    'weekday_pct',
    'days_from_first_signup']
```



Linear models that use **OLS** like **Linear Regression** sensitive to continuous variables that “clump” around an extreme value. `avg_dist`, `avg_rating_by_driver`, `avg_rating_of_driver` display this clumping. To address clumping, you can create separation in your variables by doing a `log(x+1)` or `exp()` transformation for better predictions and error estimates. Again, this **only matters for models with an estimation method sensitive to clumping**. You can see that both test and training accuracy with Logistic Regression roughly unaffected by this transformation:

```
# compare untransformed to transformed
for sets in [ColsLoglp, NewColsLoglp]:
    feature_space = DfImp[sets]
    y = DfImp['churn']
    features = feature_space.columns.tolist()
    X = feature_space.as_matrix().astype(np.float)

    # show improvement in class separation from logging
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)

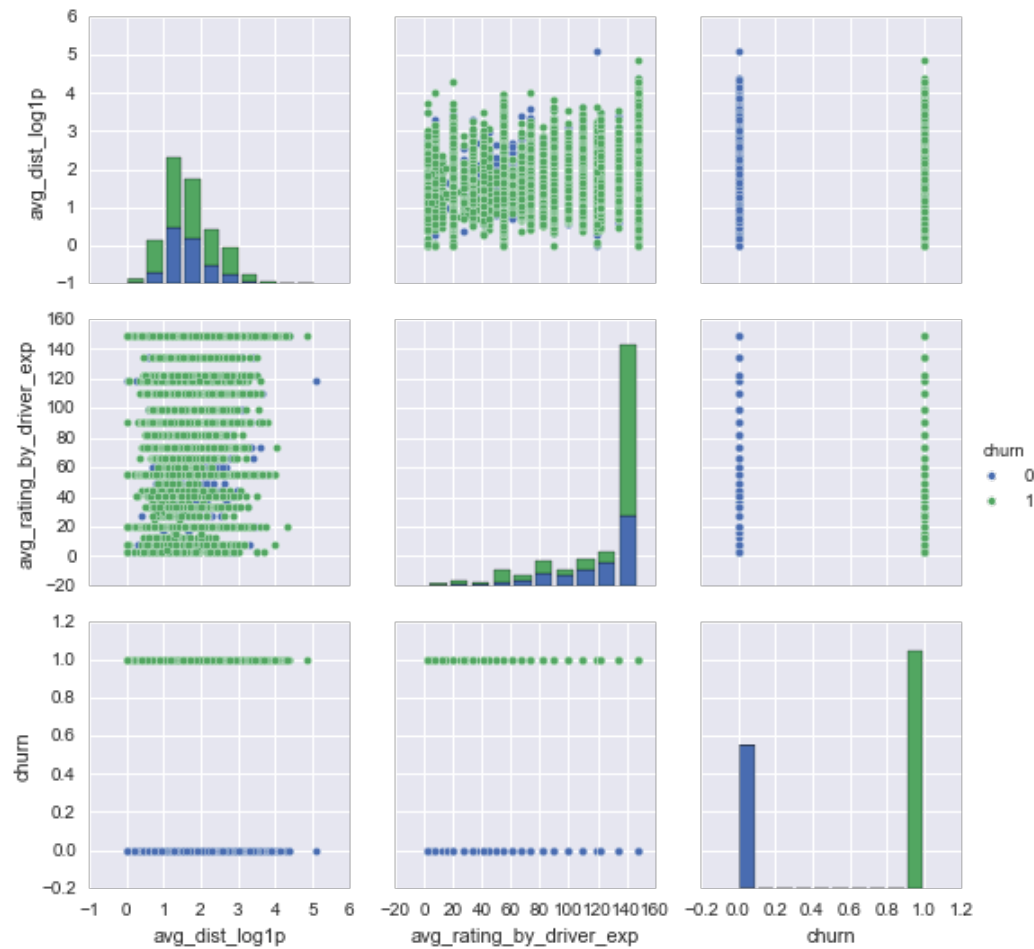
    # run logit
    model = LGT(penalty='l2', C=100)
    model.fit(X_train, y_train)
    print('Training accuracy:', model.score(X_train, y_train))
    print('Test accuracy:', model.score(X_test, y_test))

('Training accuracy:', 0.6573432835820896)
('Test accuracy:', 0.65884848484848479)
('Training accuracy:', 0.6563582089552239)
('Test accuracy:', 0.65703030303030308)
```

For machine learning models like Decision Trees (doesn’t matter at all), SVM and Neural Nets this matters much less. Instead of taking a `log()` or exponent of your distribution you standardize (convert the distribution to **standard normal**) the feature. We will discuss why in the **Feature Scaling** section. But in general, **you don’t need to do the `log()` and `exp()` transformations for most machine learning models**, it only helps linear models and measures, specifically Linear and Logistic Regression, and **Pearson Correlation**.

Let’s look at the graphs of `log` of `avg_distance` and the exponent of `avg_rating_by_driver` to see how much more dispersed these features are after the transformation:

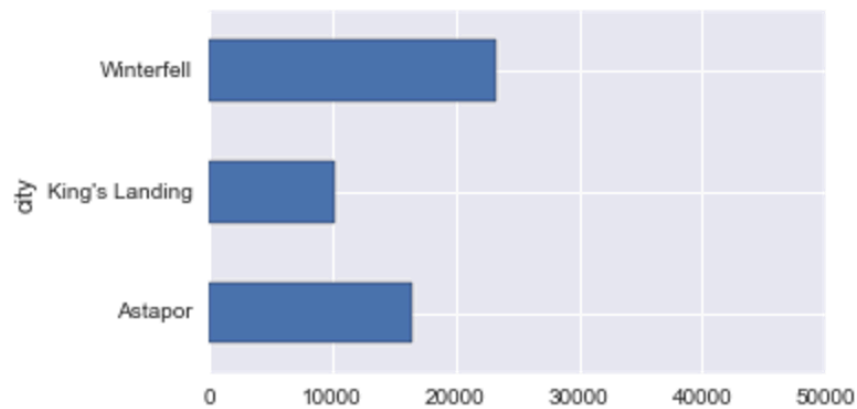
```
# visual comparison of transformation
ColsTest = [
    'avg_dist_loglp',
    'avg_rating_by_driver_exp' ]
sns.pairplot(DfImp[ColsTest + ['churn']], hue='churn')
```



It's also worth noting that `avg_dist_log1p` and `avg_rating_by_driver_exp` appear to have a persistent positive relationship. Features that **covary** tightly with each other are good candidates for **dimensionality reduction**. We'll see in the **Feature Selection** section just how tight the relationship is between these two fields.

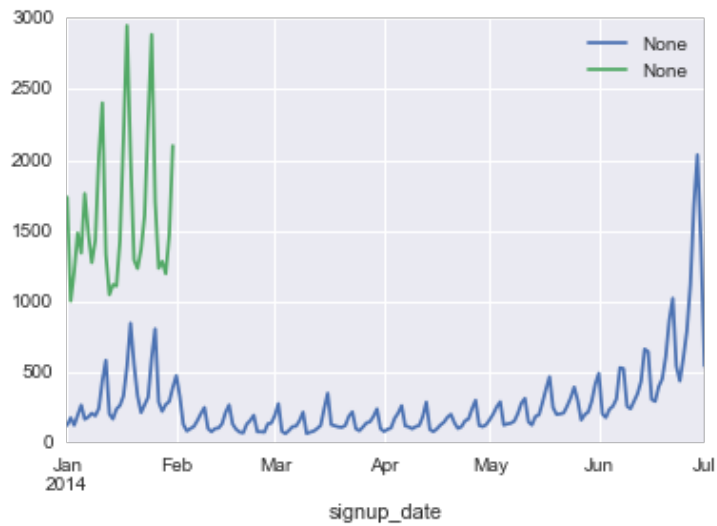
Categorical Features

Categorical features are easiest to view through bar charts with the y-axis scaled to the full size of the data set so you can eyeball the percent of the entire set it comprises.



Date Features

Dates should be plotted as line charts so you can identify trends over time, missing dates and other anomalies, like seasonality and spikes.



Feature Scaling and Other Preprocessing

Feature Scaling and Continuous Features

The two most important reasons for feature scaling are: 1) **gradient descent optimization** and 2) **model output interpretation**.

In gradient descent, if one feature is on a very small scale with a steep curve, it can prevent convergence of the algorithm because the steps by which gradient descent iterates becomes too large. Effectively, a small scale can “confuse” the algorithm. Additionally, gradient descent starts by initializing parameter value at 0. Centering features around 0 helps the algorithm do its job better. You can find more information here: http://sebastianraschka.com/Articles/2014_about_feature_scaling.html.

The second reason for feature scaling is interpreting weights on models. A classic example is in basketball where a player’s avg points per game can be an order of magnitude or two greater than his field goal percentage. So when you run a logistic regression, the weights will suggest that a player’s field goal percentage is 10x or 100x more important than points in a game in a classification exercise, but it’s really just a matter of scale.


So we start by scaling all of our continuous features:

```
# convert continuous variables to standard scalar
scaler_ss = StandardScaler()
features_cont = Df.dtypes[Df.dtypes.values == 'float64'].index.tolist()
X_cont = Df[features_cont].as_matrix().astype(np.float)
X_cont = scaler_ss.fit_transform(X_cont)
```

The **StandardScaler()** transformer in sci-kit learn converts features to the **Standard Normal Distribution**. This is a distribution with mean = 0 and standard deviation = 1.

Categorical and Boolean Features

We convert categorical features into a set of dummy or boolean features a process called **one-hot encoding, binarization or dummification**. The transformation looks like this:

state	state_num		is_ny	is_ca	is_wa
new york	1		1	0	0
california	2		0	1	0
new york	1		1	0	0
new york	1		1	0	0
california	2		0	1	0
washington	3		0	0	1
washington	3		0	0	1

The reason for this is computers don't inherently know what to do with strings. So in the background strings are encoded as integers in a single feature (state_num). The problem is integers are interpreted by their values. 3 is greater than 2 is greater than 1. The problem is 'washington' isn't inherently better or bigger than 'new york' or 'california.' So to make the values interpretable to a computer, you separate the feature into however many categories there are within it.

```
# binarize categoricals
onehot = pd.get_dummies(Df[['phone', 'city']]) # convert to binary
Df = pd.concat([Df, onehot], axis=1) # append new fields to df

# drop unneeded fields
to_drop = ['signup_date', 'last_trip_date', 'phone', 'city', 'churn']
Df = Df.drop(to_drop, axis=1)

# extract boolean
features_bool = Df.dtypes[Df.dtypes.values != 'float64'].index.tolist()
X_bool = Df[features_bool].as_matrix().astype(np.float)
```

Using Optimized Data Structures

You'll notice that while scaling and binarizing features, we also converted the feature sets to **matrices of floats**. Sci-kit learn models takes matrices and vectors as inputs because it can then perform matrix operations (linear algebra) that allow large scale operations to happen almost instantly using functions (matrix functions, dot products, cross products, etc.) and technologies (graphics cards) that are optimized for them.

After we performing all of the transformations and optimizations to booleans and categoricals separately, we rejoin the matrices and create a dataframe that represents it for future use.

```
# aggregate features
features = features_cont + features_bool
X = np.hstack([X_cont, X_bool])

# create dataframe
DfScaled = pd.DataFrame(X, columns=features)
DfScaled['churn'] = y
```

As a last step we make sure the data we want is the data we have:

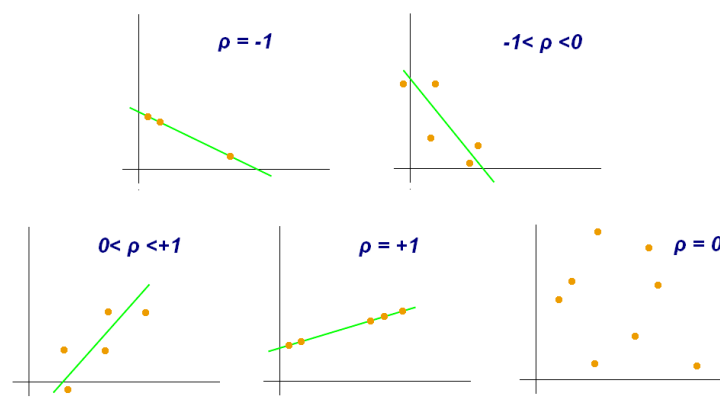
```
# look at data ranges
DfScaled.describe()
```

Feature Selection

What features are important? This mostly matters to identify factors that influence your outcome or to reduce the feature space so that the model can run faster.

Correlation

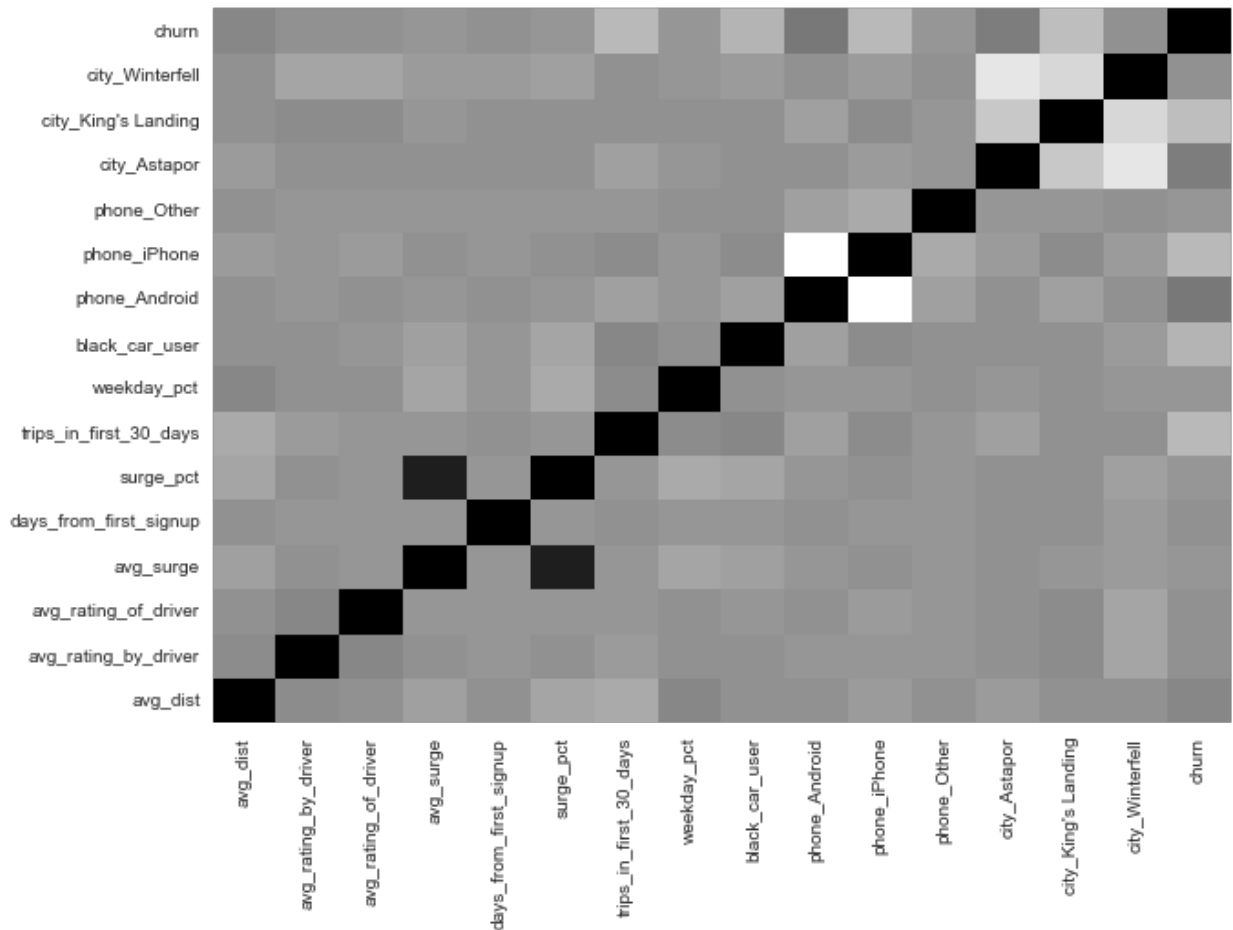
Formally, correlation is the strength of a **linear relationship** between two sets of values. A positive value means the relationship is positive. E.g. height is positively correlated with weight. But the actual value tells you **how linear** the relationship is. So if all points form a perfect line, then the relationship is very linear. Though it doesn't mean that if it is a large value (close to 1), that the feature is more important than another one; it just means the relationship is more linear/straightforward. This is important when working with linear models, but can still be instructive when working with other models. Here are some examples of what a different correlation values look like plotted:



Let's look at our correlations:

```
# correlation matrix
DfHeatmap = DfScaled.corr()
DfHeatmap
```

When looking at the above correlation matrix focus first on the **relationship between churn and other features**, then look for **features that are related to each other**. Features that are related to each other are candidates for PCA and can also crowd each other out of an analysis since they're so closely related.



In the above plot surge_pct and avg_surge are dark black because they're closely related. People only have a value for avg_surge if they've taken some rides with surge pricing (surge_pct). They are 80% correlated, so you could just create a new feature out of them. Similarly phone_Android and phone_iPhone are negatively correlated... because a phone can't be both, it is only one or the other. Other interesting observations are how dark black_car_user and city_Astapor are.

Lasso Feature Selection

By penalizing the cost function in Logistic Regression we can see which features have the biggest weights associated with them/which features are most related to Churn in Logistic Regression. First we split test and train then fit Logistic regression using varying levels of C (big C = no penalty, small C = big penalty).

```

# split into test/train
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)

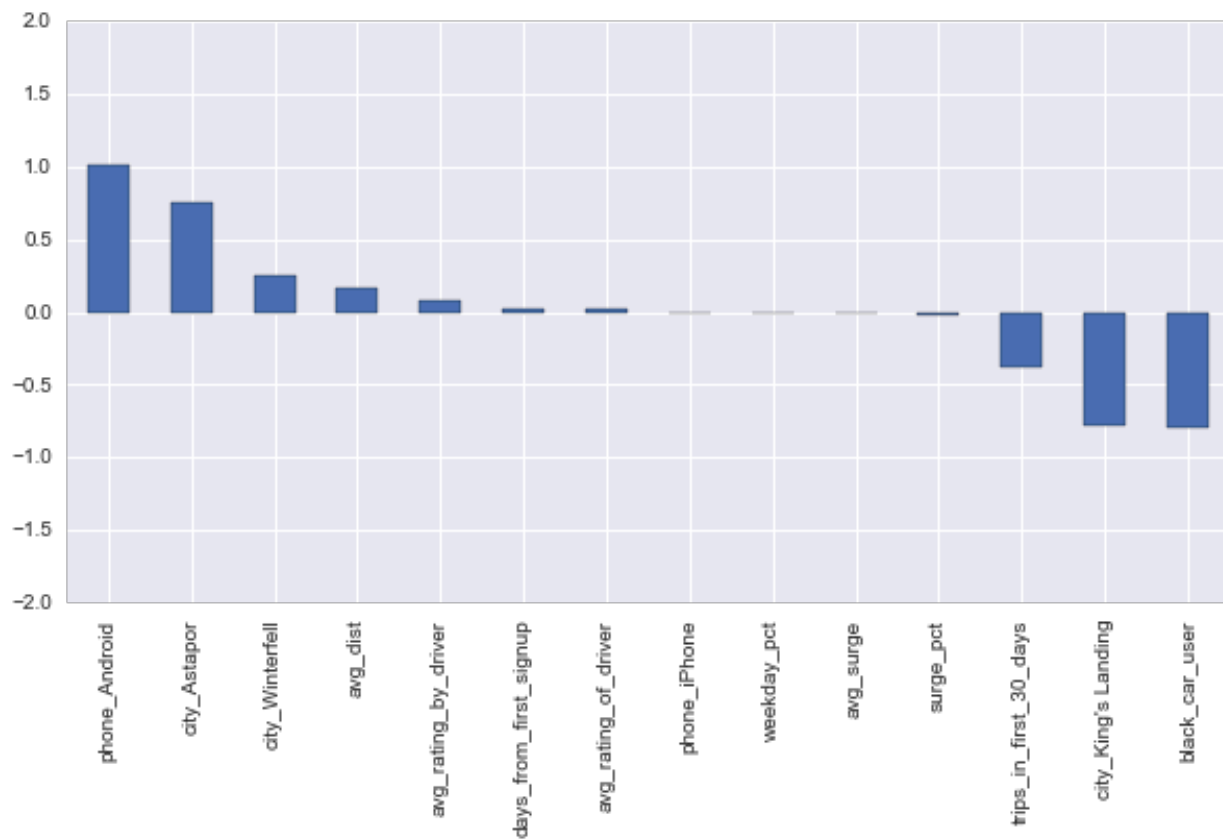
# train with l1/lasso
lr = LGT(penalty='l1', C=0.01)
lr.fit(X_train, y_train)

# plot weights
lr_coefficients = pd.Series(lr.coef_[0], index=features).sort_values(ascending=False)
plt.figure(1, figsize=(10, 5))
lr_coefficients.plot(kind='bar', ylim=(-2,2))

# accuracy
print('Training accuracy:', lr.score(X_train, y_train))
print('Test accuracy:', lr.score(X_test, y_test))

```

What we find is that boolean variables are the most indicative of churn with a linear model:

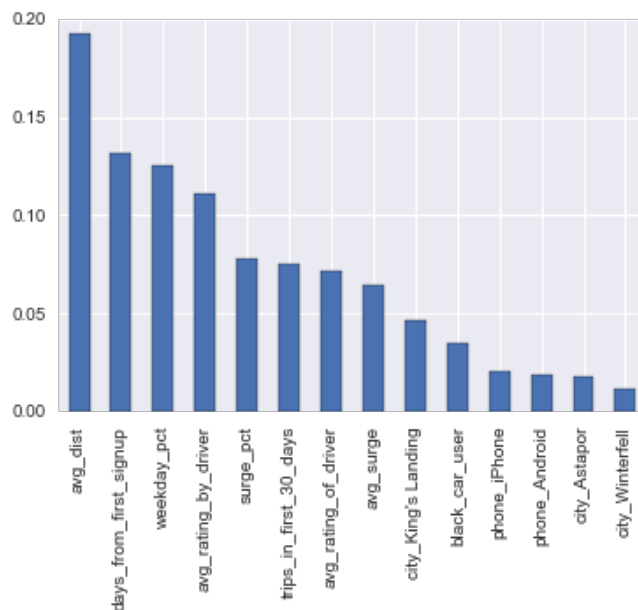


Random Forest Feature Importance

As an alternative to Lasso feature selection, we can look at Random Forests to point us to the most important features. Here, continuous variables dominate the model and binary features clearly come last:

```
forest = RF(n_estimators=250, n_jobs=-1)
forest.fit(X_train, y_train)

importances = forest.feature_importances_.tolist()
pd.Series(importances, index=features).sort_values(ascending=False).plot(kind='bar')
```



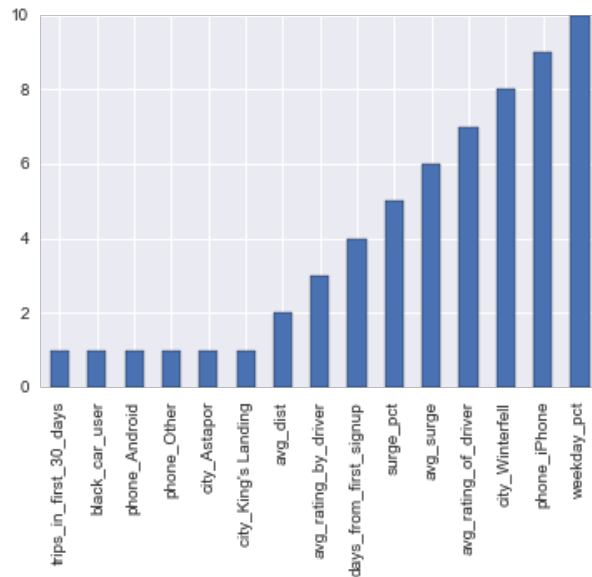
Recursive Features Elimination

Since the results of Lasso and Random Forest feature selection are near-opposites, it's worth taking an impartial method to feature selection to see what the impact of each feature is on that model. **Recursive Feature Elimination iteratively removes** the feature with the highest weight from the model, until it has a ranking of which features most matter to the model. Here it is with Logistic Regression:

```
# with logistic regression
from sklearn.feature_selection import RFE

model = LGT(penalty='l1', C=100, n_jobs=-1)
# create the RFE model and select 3 attributes
rfe = RFE(model, n_features_to_select=6, step=1)
rfe = rfe.fit(X_train, y_train)

# summarize the selection of the attributes
pd.Series(rfe.ranking_.tolist(), index=features).sort_values().plot(kind='bar')
```

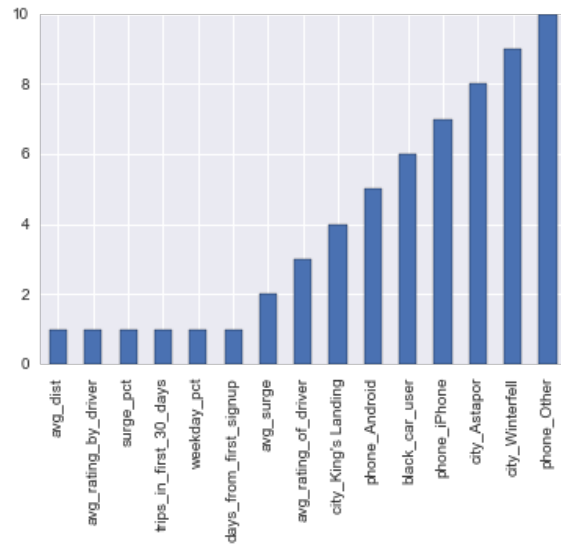


And with Random Forest:

```
# with random forest
from sklearn.feature_selection import RFE

model = RF(n_estimators=50, n_jobs=-1)
# create the RFE model and select 3 attributes
rfe = RFE(model, n_features_to_select=6, step=1)
rfe = rfe.fit(X_train, y_train)

# summarize the selection of the attributes
pd.Series(rfe.ranking_.tolist(), index=features).sort_values().plot(kind='bar')
```



The results are consistent with what we found originally—for this problem, boolean features matter for Logistic Regression and continuous features matter for Random Forest. The way I interpret this is that Logistic Regression doesn't take full advantage of the continuous variables because of the rigid assumption that they are linearly related to the **logs odds of the target variable**.

Model Horse Race

Select Models with Default Settings

We start by initializing a dictionary of models with some smart default settings. For example the activation function 'relu' for neural nets runs much faster than 'logit'. The default n_estimators value of 10 for Random Forest is usually way too small, and so on.

```
from sklearn.model_selection import cross_val_predict

# create dict of models
models_dict = {
    'Neural Net':      MLPClassifier(activation='relu'),
    'Random Forest':   RandomForestClassifier(n_estimators=100),
    'KNN':             KNeighborsClassifier(n_neighbors=11),
    'Logistic Regression': LogisticRegression(C=1000),
    'Naive Bayes':     GaussianNB(),
}
```

Fitting with Cross Validation

The function **cross_val_predict** creates a **Stratified K-Fold** cross validation object and runs test/train with each fold. We store the predictions to **predictions_dict** so we can do a class vote later. The output of this loop gives us Accuracy, AUC, Precision, Recall and F1 Score. We will go into more detail with these in the single-model example.

```
# run models and get output
predictions_dict = {}
for model in models_dict:
    %time
    y_pred = cross_val_predict(models_dict[model], X=X, y=y, cv=10, n_jobs=-1)
    predictions_dict[model] = y_pred
    print model + ' metrics:\n-----'
    print 'Accuracy: {0}'.format(str(round(metrics.accuracy_score(y, y_pred), 3)))
    print 'AUC: {0}'.format(str(round(metrics.roc_auc_score(y, y_score=y_pred), 3)))
    print 'Classification Report:\n{0}\n\n'.format(metrics.classification_report(y, y_pred))
```

Summary Model Output

KNN metrics:

Accuracy: 0.754

AUC: 0.727

Classification Report:

	precision	recall	f1-score	support
0	0.69	0.62	0.65	18804
1	0.78	0.84	0.81	31196
avg / total	0.75	0.75	0.75	50000

Accuracy = the percent of predictions that were correct = $(TP + TN) / \text{all outcomes}$

AUC = the area under an ROC curve. The roughly represents the models ability to separate two classes.

Precision = $TP / \text{Positive Class Guesses}$ = "The percent of times you were right when you guessed that the outcome would be a 1"

Recall = $TP / \# \text{ Samples that into the Positive Class}$ = "The percent of samples in the positive class you correctly identified."

F1 = the harmonic mean of Precision and Recall. Harmonic mean penalizes low values.