django-ai Documentation

Release 0.0.2.1

Rodrigo Gadea

Nov 21, 2018

Contents

Contents:

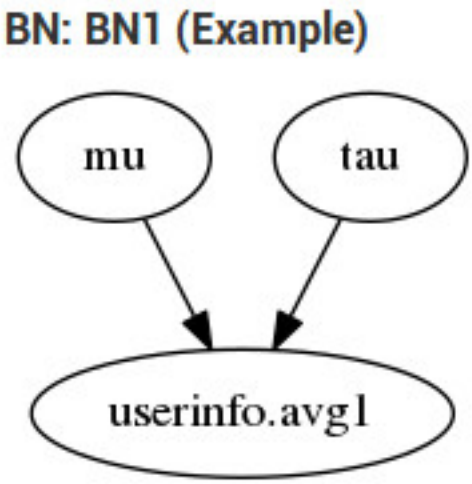
CHAPTER 1

django-ai

1.1 Artificial Intelligence for Django

`django-ai` is a collection of apps for integrating statistical models into your Django project, providing a framework so you can implement machine learning conveniently.

It integrates several libraries and engines your Django app with a set of tools so you can leverage the data generated in



your project.

1.1.1 Documentation

The full documentation is at https://django-ai.readthedocs.io or the */docs* directory for offline reading.

1.1.2 Features

- *Bayesian Networks*: Integrate Bayesian Networks through your models using the BayesPy framework.

- *Spam Filtering*: Integrate Spam Filters to your Django project using the scikit-learn framework.

See the *Introduction* section in the documentation for more information.

1.1.3 Communication Channels

- Mailing List: django-ai@googlegroups.com
- Chat: https://gitter.im/django-ai/django-ai
- GitHub: https://github.com/math-a3k/django-ai/issues
- Stack-Overflow: https://stackoverflow.com/questions/tagged/django-ai
- AI Stack Exchange: https://ai.stackexchange.com/questions/tagged/django-ai

1.1.4 Quickstart

The easiest way of trying *django-ai* is inside its package:

1. Create a virtual environment and activate it:

```
python3 -m venv django-ai_env
source django-ai_env/bin/activate
```

2. Upgrade `pip` and install `django-ai`:

```
(django-ai_env) pip install --upgrade pip
(django-ai_env) pip install django-ai
```

3. Change into the *django-ai* directory, i.e.:

```
(django-ai_env) cd django-ai_env/lib/python3.5/site-packages/django_ai
```

4. Create the migrations for the dependencies and apply them:

```
python manage.py makemigrations
python manage.py migrate
```

5. Create a superuser:

```
python manage.py createsuperuser
```

6. Start the development server and visit http://127.0.0.1:8000/admin/, look at the examples and start creating your statistical models:

```
python manage.py runserver
```

You can also clone it from the repository and install the requirements in a virtualenv:

```
git clone git@github.com:math-a3k/django-ai.git
```

and following the previous steps, install the requirements - `pip install -r requirements.txt` - in a virtual environment instead of the package.

For installing it in your project, please refer *here*.

1.1.5 Running Tests

Does the code actually work?

```
source <YOURVIRTUALENV>/bin/activate
(myenv) $ pip install -r requirements_test.txt
(myenv) $ PYTHONHASHSEED=0 python runtests.py
```

CHAPTER 2

Introduction

`django-ai` is a collection of apps for integrating statistical models into your Django project, providing a framework so you can implement machine learning conveniently.

It aims to integrate several libraries and engines supplying your Django apps with a set of tools for leveraging your project functionality with the data generated within.

The integration is done through Django models - where most of the data is modelled and stored - and with an API focused on integrating seamlessly within Django projects' best practices and patterns. This data is what will feed the statistical models and then those models will be used in the project's code to augment its utility and functionality.

The rationale of `django-ai` is to provide for each statistical model - system or technique bundled - a front-end for configuration and an API for integrating it into your code.

The front-end aims to let you choose which parts of the Django models will be used and "configure" its parameters: conveniently "state" the statistical model. Currently, it is admin-based.

Once you are "happy" with your model you can incorporate it into your code, i.e. in a view:

```
from django_ai.apps.bayesian_networks.models import BayesianNetwork

def my_view(request):
    user_classifier = BayesianNetwork.objects.get(name='User BN')
    user_class = user_classifier.predict(request.user)
    # Do something with the user classification
    return(redirect('promotions', user_class))
```

This "kind of hybrid approach" is what gives you convenience: you can state the model easily in the front-end (admin), and after you incorporate it in your code, you can maintain, improve or update it in the front-end.

By design, `django-ai` tries to take outside Django's users' request-response cycle as many calculations as possible. This means all the heavy stuff like model estimation is done once and stored. Inside the request cycle - i.e. in a view or a template - it is just regular queries to the database or operations that are "pretty straight-forward", like cluster assignment or classification.

See *Examples* for more.

`django-ai` aims to provide with 2 classes of apps or statistical models: "low level" and "high level".

"Low level" are those "basic" models or techniques, such as Bayesian Networks, Support Vector Machines, Classification and Aggregation Trees, Random Forests, Clustering algorithms, Neural Networks, etc. Those are the building blocks for the machine to learn and construct its intelligence.

"High level" are systems, they are composed from "low level" ones and provide end-to-end functionality on a certain task, such as a recommender system or a spam filter.

The `django-ai` apps will integrate the statistical models already implemented in other libraries as much as possible, using them as engines and becoming also a front-end to them for Django projects.

The primary or main integration is with Python codebases - for obvious reasons. In the future, integration with other codebases is in sight, such as `R` where the amount of statistical models implemented is the biggest and greatest or `Haskell`, where the purity of the language makes it ideal for expressing mathematical models.

This is an Introduction to the Philosophy, Design, Architecture and Roadmap of `django-ai`.

You are welcome to join the community of users and developers.

Last but not least: `django-ai` is, and will always be, Free Software (Free as in Freedom). If you can't patent Math, you can't patent Software. Did Newton hide something from you? :) Open Knowledge is better for all :)

CHAPTER 3

Installation

For installing `django-ai` in your Django project use the following steps:

1. Activate your virtual environment and then:

```
pip install django-ai
```

2. Add it to your *INSTALLED_APPS*:

```
INSTALLED_APPS = (
    ...
    # Dependencies
    'nested_admin',

    # django-ai apps
    'django_ai.base',
    'django_ai.bayesian_networks',
    'django_ai.supervised_learning',
    'django_ai.systems.spam_filtering',

    # optional but highly recommended
    'django_ai.examples',
    ...
)
```

The `django_ai.examples` is optional but it is highly recommended that you keep it as a reference.

3. Create the migrations for the dependencies and apply them:

```
python manage.py makemigrations
python manage.py migrate
```

4. Add django-ai's apps URL patterns and its dependencies:

```
urlpatterns = [
    ...
```

```
    url(r'^nested_admin/', # Dependency
        include('nested_admin.urls')),
    url(r'^django-ai/',
        include(django_ai.base.urls)),
    ...
]
```

5. Ensure that the `admin` app is enabled.
6. Ensure that your `static` serving is properly configured, if not you may have to add to your `urls.py`:

```
...
from django.conf.urls.static import static

urlpatterns = [
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

For reference, there is a working `settings.py` and `urls.py` in the source distribution for further troubleshooting if necessary.

CHAPTER 4

Available Applications in *django-ai*

The following applications are available:

4.1 `django-ai` Base

This app provides the basis for the framework.

It provides the Abstract Base Models as well as common views and template-related code.

4.1.1 Models

Abstract Base Models are documented in *API*.

4.1.2 Views

class base.views.**RunActionView**(***kwargs*)

Runs common Actions for Systems and Techniques

4.2 Bayesian Networks

This app provides Bayesian Network modelling through integrating the BayesPy framework: http://bayespy.org.

If you are not familiar with the framework, it is better at least take a glance on its excellent documentation for a better understanding on how the modelling is done.

4.2.1 Front-end

All the configuration should be done through the admin of Bayesian Networks - or more specifically, through the *change form*.

Bayesian Network

This is the main object for a Bayesian Network (BN).

It gathers all Nodes and Edges of the DAG that defines the Network.

All the results of the inference will be available here and this object is what you will be using inside the code.

The following fields are available for configuration:

Name The name of the Bayesian Network. This must be an unique identifier, meant to be used for retrieving the object (i.e. `BayesianNetwork.objects.get(name='BN 3 - Final')`)

Network Type The type of the Network. Based on this field, the internal methods of the Bayesian Network object perform different actions. Currently, there are 2 Types:

General Performs the inference with the BayesPy engine on the Bayesian Network and set the resulting object in the `engine_object` field.

Clustering Besides performing the inference with the BayesPy engine (and setting the result in the `engine_object` field), it performs tasks like cluster re-labelling, process the results and stores useful information in the `metadata` field.

It assumes that the Network topology is from a Gaussian Mixture Model with:

- One Categorical Node for clusters assigments depending on a Dirichlet Node for prior probabilities;
- One Gaussian Node for clusters means and a Wishart Node for clusters covariance matrices;
- One Observable Mixture Node for the observations.

Other topologies are not supported at the moment.

Results Storage In the case of networks which have a labelling output - such as Clustering or Classification - this sets where to store the results for convenience. It must have the following syntax: `<storage>:params`.

The following storages are available:

dmf *Django Model Field*: Saves the results to a field of a Django model. The model should be accessible by Django's Content Types framework and - **IMPORTANT**: it uses its default order provided by the model manager for storing. That ordering should be the same as the data retrieved by *Bayesian Network Node Column*, otherwise "manual" storing should be done for your situation.

Its parameters are a dotted path: `<app_label>.<model>.<field>`, i.e. `dmf:examples.UserInfo.cluster_1` will store the results to the `cluster_1` field of the `UserInfo` model.

Miscellaneous Fields

Engine Meta Iterations Runs the Inference Engine (BayesPy's VB) *N* times and picks the result with the highest likelihood. This is only useful when a Node in the Network requires random initialization (see *Custom Keywords*), as the algorithm may converge to a local optimum. Otherwise, it will repeat the result *N* times. It defaults to 1.

Engine Iterations The maximum number of iterations of the Inference Engine (BayesPy's VB update method's repeat). It defaults to 1000.

Counter Internal Counter of the Bayesian Networks meant to be used in automation. Is up to the user to increment the counter when deemed necessary. If the field `Counter Threshold` is set, when this counter reaches that Threshold, the actions in `Threshold Actions` will be run on the object's `save()` method or the evaluation can be triggered with the following method:

`BayesianNetwork.parse_and_run_threshold_actions()`

IMPORTANT: As it is up to the user when, where and how the counter is incremented, the user should take care also to avoid triggering `Threshold Actions` inside of the user's "navigation" request cycle, which may lead to hurt the user experience. For a concrete example, see *here*.

`Counter Threshold` Threshold of the Internal Counter, meant to be used in automation. If this field is not set, the `Threshold Actions` will not be triggered on the object's `save()` method.

`Threshold Actions` Actions to be ran when the Internal Counter reaches or surpasses the Counter Threshold, evaluated on model's `save()`. The actions must be specified by keywords separated by spaces. Currently, the supported keywords are:

`:recalculate` Recalculates (performs again) the inference on the Network.

`Engine Object Timestamp` This is an auto-generated field, timestamping the last inference done or *None* if not available.

`Image` This is an auto-generated field, shown at the bottom of the page. It will be updated each time a Node or an Edge is added or modified to the Network.

`Metadata` This is an internal field for storing results and information related to internal tasks (pre-processing, visualization, etc.). It is shown here for convenience as its content may be used for integrating the Bayesian Network into the application's code.

Bayesian Network Node

Each BayesPy Node in the Network is represented here.

Nodes can be either *Stochastic* or *Deterministic*.

Stochastic refers to representing a Random Variable, *Deterministic* refers to representing a transformation or function on other Nodes.

Each type of Nodes have a fields associated with it, which will need to be filled accordingly.

General Fields

`Name` The name of the Node. This must be an unique identifier inside the network. It will be used for passing the Node to other Nodes as a parameter among others.

`Node Type` If the Node is *Stochastic* or *Deterministic*. This determines which fields will be taken into account for Node creation.

Stochastic Fields

`Distribution` The Distribution of the Node

`Distribution Params` The Parameters for the Distribution of the Node. See *Node Parameters* below for more details.

The Parameters must be according to the Distribution chosen, otherwise the initialization of the BayesPy Node will fail. For a list of the Distribution Parameters see the BayesPy documentation for Stochastic Nodes.

`Is Observable` If the Random Variable is observable or not. If it is observable, then it will need to be linked to fields or callables of a Django Model where the data will be held, set in *Bayesian Network Node Column*.

Deterministic Fields

`Deterministic` The function or transformation that the Node applies.

`Deterministic Params` The Parameters for the function of the Node. See *Node Parameters* below for more details.

The Parameters must be according to the Deterministic Node chosen, otherwise the initialization of the BayesPy Node will fail. For a list of the Deterministic Parameters see the BayesPy documentation for Deterministic Nodes.

Node Parameters

The string set in the `Distribution Params` and `Deterministic Params` fields is parsed and used for initialization of BayesPy Nodes.

It is designed to be a just like the `*args` and `**kwargs` you pass to a function or method programmatically with some restrictions, described below:

`Booleans and Keywords` `True`, `False`, `None`.

`Scalars` Integers and Floats, i.e `1, -2, -0.3, 1e-06`.

`Structures` Lists and Tuples, i.e. `[1, 2], [[1e-06, 2], [3, 4]], (2, 3,), ([1, 2], [3, 4])`.

`Strings` Strings are reserved for Node names. To pass another Node as a parameter to it simply use its name. Nodes' names are resolved through Network Edges of the graph (see *Bayesian Network Edge*).

`Custom Keywords` Strings starting with `:` - i.e. `:no` are considered as "Custom Keywords" for `django-ai` and triggers different behaviours. See *Custom Keywords*.

`Functions` Functions *must be namespaced* and their arguments can be anything of the above.

In some occasions, there must be a reference to a function instead of the result of it, this is done by preceding the function with an `@`, i.e. `@bayespy.nodes.Gaussian()` will return the function object (in this case the whole class) instead of the result of it.

Due to security reasons, the allowed namespaces must be specified in a list named `DJANGO_AI_WHITELISTED_MODULES` in your settings, i.e.:

```
DJANGO_AI_WHITELISTED_MODULES = ['numpy', 'bayespy.nodes', 'scipy']
```

By default, only `numpy` and `bayespy.nodes` are enabled.

For example, the string:

```
True, 2, 1e-6, mu, numpy.ones(2), [[1,2], [3,4]], type=rect, sizes=[3, 4,], coords =␣
↪([1,2],[3,4]), func=numpy.zeros(2)
```

will be equivalent to doing programmatically:

```
MyNode(True, 2, 1e-6, mu, numpy.array([ 1.,  1.], [[1,2], [3,4]], type=rect, sizes=[3,
↪ 4,], coords = ([1,2],[3,4]), func=numpy.array([ 0.,  0.])
```

With this, a `GaussianARD` Node can be initialized with:

```
mu, tau
```

where `mu` and `tau` are parents Nodes, or for a 2D `Gaussian` Node:

```
numpy.ones(2), numpy.zeros(2)
```

Custom Keywords

Node parameters' strings starting with `:` are considered *Custom Keywords*, they should be used at an `*arg` level and their meaning or behaviour triggered is described below:

`:noplates` Triggers the deletion of the `plates` keyword argument. Use this when you do not want a keyword argument to be set automatically. Currently, `plates` is only set automatically for Stochastic Observable Nodes when it is not specified and it is set to the "shape" of the data being observed. In some types of networks this can interfere with *BayesPy* plates propagation. To avoid this, use `:noplates` in the Node's parameters.

`:ifr` Triggers *Initialize from Random* in the Node's engine object.

`:dl|<NODE_NAME>` Uses the Data Length of of Node `NODE_NAME`. Meant to be used in plates, i.e. `plates=(:dl|Y, )`

Visualization

`Graph Interval` (*Stochastic only*) Depending on the Distribution, a graphic may be available. This is the graphing interval, separated by a comma and a space, i.e. "`-10, 20`".

`Image` (*Stochastic only*) This is an auto-generated field, once the inference is run on the network, if it is available, an image with the graph of the distribution of the Random Variable will be stored here and shown at the bottom of the page.

Timestamps

`Engine Object Timestamp` The Timestamp of the BayesPy Node creation.

`Engine Inferred Object Timestamp` The Timestamp of the last inference on the Node or the network.

Bayesian Network Node Column

In the case of Stochastic Observable Nodes, an inline will be displayed for setting the columns / axis / dimensions that will represent the observations of the Random Variable of the Node.

There is no restrictions on number of columns nor they should be on the same Django model, only that they must contain the same amount of records / size.

The following fields are shown:

`Reference Model` The Django Model that will held the data.

`Reference Column` The name of the field or attribute in the Django model that will held the data.

`Position` The ordering of the columns, set automatically by the nested inline.

Bayesian Network Edge

Each Edge between Nodes in the Direct Acyclic Graph of the Bayesian Network is represented here.

Edges are necessary for resolving dependencies between nodes, i.e. if Node takes another Node as a parameter, there must be an Edge between the Nodes so the Child is able to access its Parents.

The following fields are shown:

Parent The "From" Node.

Child The "To" Node.

Description A brief description of the Edge (i.e. "`mu -> tau`").

Actions

The main are:

Run inference on the network This will run the Inference on the current state of the network.

It will initialize or create all the BayesPy Nodes, initialize the Inference Engine of BayesPy, perform the inference with it and the appropriate tasks corresponding to the *Network Type*. Once is run, it will save all the results in the Bayesian Network object and the corresponding in each Node, generating the Node image where corresponds and updating the timestamps. See the API section for accesing the results.

Reset inference on the network This will reset (set to *None*) all the engine- and inference-related fields in the network.

Re-initialize the random number generator This will reinitialize Python's random number generator. For unknown reasons yet, sometimes the Inference Engine gets stuck, re-initializing the RNG and resetting the inference may solve the issue without restarting the server.

4.2.2 API

For integrating the objects into your code, you simply have to import the Django model whenever deemed necessary and get the network you want to use:

```
from django_ai.models.bayesian_networks import BayesianNetwork

bn = BayesianNetwork.objects.get(name="<NAME-OF-MY-BN>")
```

If the network is inferred, the results of it - the `VB` object - is stored in the `engine_object` field. This is a BayesPy object which you can use at your will:

```
Q = bn.engine_object
mu = Q['mu'].get_moments()[1]
tau = Q['tau'].get_moments()[1]
sigma = sqrt(1 / tau)
if (request.user.avg1 < mu - 2 * sigma or
    request.user.avg1 > mu + 2 * sigma):
    print("Hmmm... the user seems to be atypical on avg1, I shall do something")
```

You can perform all the Actions on the Network with the following methods of the `BayesianNetwork` objects:

class bayesian_networks.models.**BayesianNetwork**(**args*, ***kwargs*)

Main object of a Bayesian Network.

It gathers all Nodes and Edges of the DAG that defines the Network and provides an interface for performing and resetting the inference and related objects.

See base.StatisticalTechnique for the fields and methods already included.

get_engine_object(*reconstruct=False*, *propagate=True*, *save=True*)

Constructs the Engine Objects of all Nodes in the Network for initializing the Inference Engine of the Network.

This is the method that should be called for initializing the EOs of the Nodes, as it handle the dependencies correctly and then propagate them to the Nodes' objects.

CAVEAT: You might have to call 'node.refresh_from_db()' if for some reason the Nodes are already retrieved before this method is ran.

perform_inference(*iters=None*, *recalculate=False*, *save=True*)
Retrieves the Engine Object of the Network, performs the inference and propagates the results to the Nodes.

reset_inference(*save=True*)
Resets the Engine Object and timestamp from the Network (the Network object itself and all the Nodes objects in it)

If you want to do things programmatically, you should see the migrations of the `examples` app:

`examples.migrations.0004_bn_example.`**create_bn1_example**(*apps*, *schema_editor*)
Create a Bayesian Network from the scratch.

`examples.migrations.0006_clustering_bn_example.`**create_clustering_bn_example**(*apps*, *schema_editor*)
Create a Bayesian Network from the scratch.

and take a look at the `tests`.

4.3 Supervised Learning

This app provides Supervised Learning techniques for integrating them into systems or directly to your code.

From an API point of view, each technique is a particular implementation of *Supervised Learning Technique*.

4.3.1 Support Vector Machines (SVM)

Support Vector Machines are provided by integrating the *scikit-learn* framework: http://scikit-learn.org.

If you are not familiar with the framework, it is better at least take a glance on its excellent documentation for the technique for a better understanding on how the modelling is done.

An example of integrating SVM into a system can be found in *Spam Filtering with SVM (Example 3)*.

SVM for Classification

All the configuration can be done through the admin of Support Vector Machines for Classification - or more specifically, through the *change form*.

The following fields are available for configuration:

General

General fields (like `Name`) and Miscellanous are documented in the *Statistical Model API*.

This technique extends it with the following field:

`SVC.`**image Image**
Auto-generated Image if available

The implementation uses *scikit-learn* as Engine, there is no need of setting more than 1 `Engine Meta Iterations`.

Model Parameters

`SVC`.**`kernel SVM Kernel`**
Kernel to be used in the SVM. If none is given, RBF will be used.

`SVC`.**`penalty_parameter Penalty parameter (C) of the error term.`**
Penalty parameter (C) of the error term.

`SVC`.**`kernel_poly_degree Polynomial Kernel degree`**
Degree of the Polynomial Kernel function. Ignored by all other kernels.

`SVC`.**`kernel_coefficient Kernel coefficient`**
Kernel coefficient for RBF, Polynomial and Sigmoid. Leave blank "for automatic" (1/n_features will be used)

`SVC`.**`kernel_independent_term Kernel Independent Term`**
Independent term in kernel function. It is only significant in Polynomial and Sigmoid kernels.

`SVC`.**`class_weight Class Weight`**
Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

Implementation Parameters

`SVC`.**`decision_function_shape Decision Function Shape`**
Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

`SVC`.**`estimate_probability Estimate Probability?`**
Whether to enable probability estimates. This will slow model fitting.

`SVC`.**`use_shrinking Use Shrinking Heuristic?`**
Whether to use the shrinking heuristic.

`SVC`.**`tolerance Tolerance`**
Tolerance for stopping criterion.

`SVC`.**`cache_size Kernel Cache Size (MB)`**
Specify the size of the kernel cache (in MB).

`SVC`.**`random_seed`**
The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

`SVC`.**`verbose Be Verbose?`**
Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

4.4 Spam Filtering

This system provides Spam Filtering through integrating the *scikit-learn* framework: http://scikit-learn.org.

It provides a pluggable filter for any Django model that is subject to Text Spam.

An example of implementing a Spam Filter into a project can be found in *Spam Filtering with SVM (Example 3)*.

4.4.1 Spam Filter

This is the main object for the Spam Filtering System.

class systems.spam_filtering.models.**SpamFilter**(**args*, ***kwargs*)
Main object for the Spam Filtering System.

All the configuration can be done through the admin of Spam Filters - or more specifically, through the *change form*.

Front-End

General

General fields (like `Name`) and Miscellanous are documented in the *Statistical Model API*.

The implementation uses *scikit-learn* as Engine, there is no need of setting more than 1 `Engine Meta Iterations`.

Spammable Model

A Spammable Model is a Django model which inherits from the *IsSpammable* Abstract Model (discussed below) for convenience of incorparting the model to all the functionality in the Spam Filtering cycle.

SpamFilter.**spam_model_is_enabled Use a Spammable Model?**
Whether to use a Spammable Model as a data source

SpamFilter.**spam_model_model Spammable Django Model**
"IsSpammable-Django Model" to be used with the Spam Filter (in the "app_label.model" format, i.e. "examples.CommentOfMySite")

If you choose not to use an Spammable Model, you can specify where the data is held (Spammable Content and Labels) via the Data Columns and Labels Column sections.

Classifier

The Classifier model to be used for discerning the Spam.

Any implementation of a *Supervised Learning Technique* using a *scikit-learn* classifier will work.

SpamFilter.**classifier Classifier to be used in the System**
Classifier to be used in the System, in the "app_label.model|name" format, i.e. "supervised_learning.SVC|My SVM"

Cross Validation

Cross Validation (CV) will be used as the perfomance estimation of the Spam Filter. The reported estimation will be the mean and the 2 standard deviations interval of the metrics evaluated in each CV fold.

CV is done with the *scikit-learn* engine, more general information is available here and here is detailed about the available metrics.

SpamFilter.**cv_is_enabled Enable Cross Validation?**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

SpamFilter.**cv_folds Cross Validation Folds**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

SpamFilter.**cv_metric Cross Validation Metric**
Metric to be evaluated in Cross Validation

Pre-Training

Pre-training refers to providing the model with "initial" data, as "initializating" the model. See *Spam Filter Pre-Training* for more details.

SpamFilter.**pretraining**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Bag of Words Representation

The Bag of Words representation (BoW) is a suitable representation for many Natural Language Processing problems - such as text classification.

If it is not enabled, the Spam Filter will use the *UTF-8 code point representation* for the corpus: each character is represented on an axis and its value is its UTF-8 code, i.e. `Hola!HOLA!` will be represented as (72, 111, 108, 97, 33, 72, 79, 76, 65, 33), and the input dimensionality will be the maximum length of the texts in the corpus.

For more information on the transformation, see the *Spam Filtering with SVM (Example 3)* and the Engine documentation.

SpamFilter.**bow_is_enabled Enable Bag of Words representation?**
Enable Bag of Words transformation

SpamFilter.**bow_use_tf_idf (BoW) Use TF-IDF transformation?**
Use the TF-IDF transformation?

SpamFilter.**bow_analyzer (BoW) Analyzer**
Whether the feature should be made of word or character n-grams. Option 'Chars in W-B' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.'

SpamFilter.**bow_ngram_range_min (BoW) n-gram Range - Min**
The lower boundary of the range of n-values for different n-grams to be extracted. All value of n such that min_n <= n <= max_n will be used.

SpamFilter.**bow_ngram_range_max (BoW) n-gram Range - Max**
The upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

SpamFilter.**bow_max_df (BoW) Maximum Document Frequency**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

SpamFilter.**bow_min_df (BoW) Minimum Document Frequency**
When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

SpamFilter.**bow_max_features (BoW) Maximum Features**
If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus. This parameter is ignored if vocabulary is not None.

Bag of Words Transformation - Miscellanous

SpamFilter.**bow_binary (BoW) Use Binary representation?**
If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

SpamFilter.**bow_enconding (BoW) Encoding**
Encoding to be used to decode the corpus

SpamFilter.**bow_decode_error (BoW) Decode Error**
Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given encoding. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.'

SpamFilter.**bow_strip_accents (BoW) Strip Accents**
Remove accents during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

SpamFilter.**bow_stop_words (BoW) Stop Words**
If 'english', a built-in stop word list for English is used. If a comma-separated string, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if analyzer == ´word´. If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.'

SpamFilter.**bow_vocabulary (BoW) Vocabulary**
A Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix. If not given, a vocabulary is determined from the input documents. Indices in the mapping should not be repeated and should not have any gap between 0 and the largest index.

API

SpamFilter extends the *Supervised Learning Technique* in several ways.

Engine-related

The Engine's Vectorizer is available / stored in a *PickleField*:

SpamFilter.**engine_object_vectorizer**
Engine Object Vectorizer

and initialized / retrieved with:

SpamFilter.**get_engine_object_vectorizer**(*reconstruct=False*, *save=True*)
Retrieves / Initializes the Engine's Vectorizer and transforms the data making it available in the *self.engine_object_data* field.

returning an instance of CountVectorizer or TdifVectorizer according to the *Spam Filter's configuration*.

The BoW-transformed data is available / stored also in a *PickleField*:

SpamFilter.**engine_object_data**
Engine Object Data

which can be retrieved or reconstructed via

SpamFilter.**get_engine_object_data**(*reconstruct=False*, *save=True*)
Retrieves / Reconstructs the BoW representation of the data.

Classification is done by the `predict` method - as expected - but with the caveat of taking a list of observations (texts) as its argument:

SpamFilter.**predict**(*texts*)
Classifies a list of observations

```
from django_aisystems.spam_filtering.models import SpamFilter

sf = SpamFilter.objects.get(name="<NAME-OF-MY-SF>")
sf.predict(["Buy Viagra Online"])
sf.predict(["Buy Cialis Online", "Oh, loved your article!"])
```

4.4.2 IsSpammable

IsSpammable is a Django Abstract Model (AM) meant to give convenience in the Spam Filtering cycle.

The AM provides the fields, options and `.save()` method to attach the model to a Spam Filter.

Once attached to a Spam Filter, the data held in the Django model will be used for training the Filter and the Filter will be used to classify new data created in the model (on `.save()` if the Spam Filter is inferred).

class systems.spam_filtering.models.**IsSpammable**(**args*, ***kwargs*)
This Abstract Model (AM) is meant to be used in Django models which may recieve Spam.

Usage:

- Make your model inherit from this AM.
- Set the SPAM_FILTER constant to the name of the Spam Filter object you would like to use
- Set the SPAMMABLE_FIELD to the name of the field which stores the content.
- Example:

```
class CommentsOfMySite(IsSpammable):
    SPAM_FILTER = "Comment Spam Filter"
    SPAMMABLE_FIELD = "comment"
    ... # The rest of your code
```

Fields and Settings

IsSpammable.**SPAMMABLE_FIELD = None**
Name of the field which stores the Spammable Content

IsSpammable.**SPAM_LABEL_FIELD = 'is_spam'**
Name of the field which stores the Spam labels

IsSpammable.**SPAM_FILTER = None**
Name of the Spam Filter object to be used

IsSpammable.**is_spam Is Spam?**
If the object is Spam - Label of the Object

IsSpammable.**is_misclassified Is Misclassified?**
If the object has been misclassified by the Spam Filter - useful for some algorithms and for understanding the filter

IsSpammable.**is_revised Is Revised?**
If the object classification has been revised by a Human - Need for proper training and automation

Usage

- Make your model inherit from this AM.
- Choose the Spam Filter to be attached by seting the SPAM_FILTER constant to the name of the Spam Filter object. you would like to use
- Set the SPAMMABLE_FIELD constant to the name of the field which stores the content.
- Make and run migrations.

Example

```
class CommentsOfMySite(IsSpammable):
    SPAM_FILTER = "Comment Spam Filter"
    SPAMMABLE_FIELD = "comment"
    ... # The rest of your code
```

Other Considerations

Technically, what makes a Django model "pluggable" into a Spam Filter as a source of data for training are:

- SPAMMABLE_FIELD constant which defines the where is the content
- SPAM_LABEL_FIELD constant which defines the field where the label is stored - defaulted to is_spam.
- A *NullBooleanField* to store the labels of the objects.

If you do not want ot inherit from the AM, any model with these three defined will work as an Spammable Model in the Spam Filter setup. The only pending thing for completing the systmes is the automation of classification of new objects.

4.4.3 Spam Filter Pre-Training

Pre-training refers to providing the model with other data, "external" data, as an initialization. That data is incorporated into the training dataset of the model.

SpamFilterPreTraining is a Django Abstract Model (AM) meant to give convenience in pre-training the Spam Filter.

class systems.spam_filtering.models.**SpamFilterPreTraining**(**args*, ***kwargs*)
Abstract Model for pre-training Spam Filters. Subclass this Model for incorporating datasets into the training of a Spam Filter (the subclass must be set in the Spam Filter's pretraining field).

Usage

- Create a Django Model that inherits from *SpamFilterPreTraining*
- Make and run migrations

- Import data to the Django Model
- Set the Spam Filter pre-training field to use the pre-training model

Example

```
class SFPTEnron(SpamFilterPreTraining):

    class Meta:
        verbose_name = "Spam Filter Pre-Training: Enron Email Data"
        verbose_name_plural = "Spam Filter Pre-Training: Enron Emails Data"
```

examples.migrations.0015_sfptenron_sfptyoutube.**download_and_process_pretrain_data_files**(*app*
sch

Forward Operation: Downloads if neccesary the sample data and populates Pre-Train Models.

Other Considerations

Technically, what makes a Django model "pluggable" into a Spam Filter as a source of pre-training are the `content` and `is_spam` fields, or the `SPAMMABLE_FIELD` and `SPAM_LABEL_FIELD` constants defined in the class pointing to Text or Char field and a Boolean field respectively.

If you do not want to inherit, define either or both in your Django Model and it will be "pluggable" as a pre-training dataset.

4.5 Examples

This app contains sample data and example objects of the apps of `django-ai`.

4.5.1 UserInfo Model

This is a generated Model that mimics information about the users of a Django project. It is common for many web applications to record metrics about the usage of the application to identify patterns besides the information of the user available. It will provide data for the statistical modelling.

It is mainly populated in the following way:

examples.migrations.0003_populate_userinfo.**populate_userinfos**(*apps*,
schema_editor)

4.5.2 Bayesian Network Example 1

The goal of this example is to show how to construct a very simple Bayesian Network for modelling the mean and the precision of a metric recorded for each user that we will assume that is normally distributed.

With this, the application can decide actions for the users based on the "position" of its metric in the population. Although this particular application can be done with point estimates, the objective here is to show the modelling.

This is based on the example provided in BayesPy's Quickstart, where you can find all the math details of the model.

Once the inference is ran, you can do something like:

```
from django_ai.models.bayesian_networks import BayesianNetwork

bn = BayesianNetwork.objects.get(name="BN1 (Example)")
Q = bn.engine_object
mu = Q['mu'].get_moments()[0]
tau = Q['tau'].get_moments()[0]
sigma = sqrt(1 / tau)
if (request.user.avg1 < mu - 2 * sigma or
    request.user.avg1 > mu + 2 * sigma):
    print("Hmmm... the user seems to be atypical on avg1, I shall do something")
```

4.5.3 Clustering with Bayesian Networks (Example 2)

The goal of this example is to show how to use unsupervised learning (clustering) to segmentate your users based on metrics of usage patterns.

For this, we will discuss the SmartDjango site, which is an example biased - but not limited - towards applications of e-commerce / shopping and content-based sites - like news, blogging, etc.

It can also be thought as a piece of a recommendation system, where the recommendations are based on what "similar" users - in the same group - do.

Recording the metrics

Recording metrics is application-specific, depends on your case: how your data is already "organized" or modelled, what you are trying to measure, your goals and the tools' availability.

In this case, the SmartDjango company knows - from their experience building and running the site over the time - that there are different groups of users which consume the application in different ways: among all the pages of the site, there are pages (content) which are not equally appealing to all the users.

They want to learn how these groups "are" in order to provide a more personalized experience to engage users - like different templates, filtering of items - or at a higher level - promotions, content, etc.

As an initial approach, starting from the scratch, the measuring of usage patterns of users should be done with the simplest metrics: how much time a user spends on a interested page (on average), and how many times does the user visits it.

The SmartDjango site is small and has about 2 hundreds of different pages (different urls for items or content items), so 400 metrics would have to be recorded for each user.

Each metric represents a dimension in the data that will feed the model, so each user usage pattern would be represented as an *R^400* point or observation. This is unsuitable for the models or techinques that we are using in this example - a Bayesian Network of Gaussian Mixtures - because they are affected by the Curse of Dimensionality (you should read this explanation for Machine Learning).

A solution for this is recording the metrics at a higher level of aggregation, instead of a metric for each page, collect for different groups / categories / types of pages. This way, the dimensionality of the input is reduced to a "useful space" for the model.

This showcase the need for "aligning" the model and the data in order to have a succesful application of the learning into the project. Having chosen a model for a goal, the data must in line with it so the solution it produces is the best within its scope or limitations - every model trades generality for acurracy in some way, for getting better in particular. This model works - it has been already proven that it will produce optimal results (in some sense) given cerntain conditions (characteristics of the input / data), having them "aligned" is what makes the best out of the tool.

If the SmartDjango company was a concrete news or blogging site, their interested pages would be the news or posts, which are usually already categorized with sections like "Sports" (A), "Tech" (B), "Art" (C), "Culture" (D), etc. or the "main" tag of the post.

In the case of a concrete shopping site, their interested pages could be the categories of their products, like "Shoes" (A), "Laptops" (B), "Bags" (C), etc.

For other content-based sites, the categories usually "emerge naturally", like in music or movies sites they could be the genres. In other cases, you may have to categorize them according to your goals.

In this case, SmartDjango has categorized their interested pages according to their role, resulting in 10 categories or types of pages - "A" ... "J" - resulting in 10 metrics of the form:

avg_time_pages_X Average Time Spent on Pages of type X (Seconds).

visits_pages_X Amount of Visits on Pages of type X (Hits).

(with *X* = "A", ..., "J")

examples.metrics.**metric_visits_and_avg_time_on_pages**(*data*)
Updates the average time on pages and its amount of visits

For implementing the metrics recording, SmartDjango had to update both front-end and back-end parts of the application.

In the front-end, the base template of the interested pages was updated to include the necessary Javascript to measure the time spent on the page and then unobtrusively POST it to the back-end, where the implemented Metrics Pipeline executes all the code that calculates and store the metrics away from the user's "navigation" request cycle:

examples.views.**process_metrics**(*request*, *verbose=True*)
Minimal implementation of a Metricps Pipeline.

The CSRF exemption is because it is highly unlikely that an external site posts values to our localhost to mess with our AI :)

Realizing that the most visited ones are from type A, SmartDjango decided to go first on a smaller step: for the first building block of the system the focus should be put on "A"-pages and then, the rest. Therefore, another metric should be recorded:

avg_time_pages Average Time Spent on Pages, independently of their type (Seconds).

visits_pages Amount of Visits on Pages, independently of their type (Hits).

examples.metrics.**metric_visits_and_avg_time_on_pages**(*data*)
Updates the average time on pages and its amount of visits

Constructing the model

The reason for choosing recording averages is that they have the "nice" property of being well modelled with Gaussian (Normal) variables - even in the "odd" cases when it is not normally distributed, it takes a factible amount of measurements in this case - time spent by users - for the approximation to be "good".

So, given that we will be dealing with averages, we will use a model that is geared towards that.

In order to find the groups (clusters) of users which share a common usage pattern, a mixture of Normal variables will be used with only 2 of the recorded metrics: `avg_time_pages` and `avg_time_pages_a`. Note that this metrics are not independent, a "hit" on a Page of type A will also make a "hit" on Pages.

Once the groups have been identified, you can take actions based on their belonging and even find more patterns inside them using "extra information" (other metrics gathered not included for this model).

This is an adaptation of the BayesPy Gaussian Mixture Model Example, where you can find more details about it, in this case there will be also 4 clusters, which can be briefly characterized by:

- A first group that stays briefly on the site, and does not care about pages of type A, they stay short independently of the page type.
- Two "opposite" "central" groups: one will stay almost the same time on pages of type A while their interest vary on other pages whereas the other has a fixed interest on other pages and a varying degree on A-types.
- A fourth group which stays longer in the site with a negative correlation: the higher they stay on pages of type A, the shorter they stay on other pages - and vice-versa.

(if interest is well measured by time spent on it :).

You can see the details of its construction in the following migration:

`examples.migrations.0005_add_avg_times_clusters.`**`populate_avg_times`**(*apps*, *schema_editor*)

For implementing the model, the `Network type` is set to "Clustering" in the Bayesian Network object and the network topology can be briefly described as:

`alpha` The prior for the probability of assignment to a cluster (group).

`Z` The belonging to a cluster (group).

`mu and Lambda` The mean vectors (`mu`) and precision matrices (`Lambda`) of the Gaussian clusters.

`Y` The mixture itself that observes the data from the users.

After the inference is performed, the model should identify the 4 real clusters from the 10 initial ones set in the uninformative priors of the models (see the nodes parameters in the admin interface).

This showcase how this technique chooses the number of clusters.

A problem with this technique is relying on random initialization (Variational Bayes engine) and the algorithm may converge to local optimums which are not the global optima.

For avoiding this, the `Engine Meta Iterations` in the Bayesian Network object (see *Bayesian Network*) is set to 15. This will repeat the inference 15 times from different states of the random number generator and choose the one with the highest likelihood. Also, `django-ai` conveniently takes care of doing all the re-labelling required between inferences so you can compare the results and automate the inference and the actions you might take according to the results.

You can watch this effect by running the inference several times from the admin's interface and tweaking the parameter.

By design, `django-ai` tries to take outside Django's request cycle as many calculations as possible. This means all the heavy stuff like model estimation is done once and stored. Inside the request cycle - i.e. in a view or a template - it is just regular queries to the database or operations that are "pretty straight-forward", like cluster assignment.

So, performance-wise, it doesn't matter if you choose 1000 meta iterations and takes an hour to complete them.

Also note that this model (Gaussians Mixture) can't "discern" when the clusters are overlapped - like the "central" "opposite" ones. For those users, it will be highly unlikely that they will be assigned to the right ones.

Once you are confident with the results, they will be stored in the `cluster_1` field of the `UserInfo` model (as it was set in the `Results Storage` field of the Bayesian Network object - see *Bayesian Networks*). Note that the `_1` in the model field is to emphasize that you may end up with several clusters for the user with different metrics and it can be used as an input for other networks or techniques.

This way, they will be efficiently available in all your views and templates, so you can do things like:

```
def my_view(request):
    ...
    if request.user.user_info.main_group == "A":
        return(redirect('/deals/A'))
    # or
    products = Products.objects.filter_for_group(
```

```
                                request.user.user_info.cluster_1)
    ...
```

Automation

After a while, many new users come and also usage patterns may change, resulting in a never-ending learning process. How to automate this dependes on the case, there are 2 main intended ways:

1. Through the BN internal counter and its threshold, which will be used in this case.
2. Through scheduled and background tasks - the preferred way.

Each time a metric is posted (see *above*) the BN counter is incremented by 1, and each time a new user is created the counter is incremented by 5. Once the counter reaches its threshold - 10 - it triggers an update of the model: it recalculates all with the new data and re-assigns the users to the new results.

These numbers are arbitrary and are intended to showcase the process. Setting the Threshold, when and how to increment the counter depends on your case. The logic here is "when 'enough' new data has arrived or the data have changed 'significantly', recalculate the model".

Also note that the Counter Threshold might trigger the update **inside** the user's "navigation" request cycle, as when creating a new user. You should realize that the process is happening when you experience the delay in the browser. This is intentional, to show what you **SHOULDN'T** do:

`examples.views.`**`new_user`**`(`*request*`)`
Mimmics creating a user by creating an UserInfo object.

In the case of processing metrics, the same code does not give any problems to the user experience because it is outside the "navigation" request cycle (see its source *above*). Try commenting out the update method in `new_user` and everything will go smoothly.

You can disable this behaviour by setting the Counter Threshold field to nothing in the admin.

Using scheduled and background tasks is the preferred way because it avoids completly the chance of messing with user's "navigation" request cycle, which may end up being detrimental to the user experience.

You can do this with apps like Celery.

Updating the counter may be just a query to the database and may be not worthy of the overhead of a background task, but the model inference is indeed what you want to schedule, i.e. at midnight:

```
@periodic_task(
    run_every=(crontab(minute=0, hour=0)),
    name="recalculate_bn_if_necessary",
    ignore_result=True
)
def recalculate_bn_if_necessaryl(bn_name, threshold):
    """
    Recalculates the model of a BN if the data has changed enough
    """
    bn = BayesianNetwork.objects.get(name=bn_name)
    if bn.counter >= your_threshold:
        bn.perform_inference(recalculate=True)
        bn.counter = 0
        bn.save()
        logger.info(
            "BN {} had a counter of {} and has been recalculated".format(
                bn.name, bn.counter)
        )
```

(You may also want to use django-celery-beat).

You can opt for no automation at all and "manually" recalculate all the model through the admin when deemed necessary. This may be suitable for the beginning, but with the adequate tuning, you can build and incorporate an autonomous system that constantly learns from the data into your application.

For the cases where there is not possible or feasible to install an app like `Celery`, with the mentioned caveats you can implement the functionality using the internal counters.

Other Considerations

Each time a model recalculation is done, you may take a look at the bottom of the admin page of the BN where the graphic of the current model is shown along with clusters table to monitor it (you may need to reload the page :).

You might find that the technique - Bayesian Gaussian Mixture Model for Clustering implemented via the BayesPy's engine - is "unstable": it produces different results when the data changes a bit - and even when it doesn't change.

This poses a problem for automation and for an AI system, as decisions are taken based on labels it produces. If the labels change dramatically, those decisions (routing, filtering, content, etc.) may end up being meaningless.

There are a variety of reasons of why this happens on two different levels - the model and the implementation - from which we will review three of them.

First, the model itself features automatic selection of the number of clusters in the data.

Selecting or determining the number of clusters is a key problem in Unsupervised Learning and not an easy one. This technique "selects automatically" that number by starting from a maximum number - set in the nodes parameters of the priors and hyperparameters - and after fitting the model, the ones that have "changed" from its "initial state" are the number of clusters in the data. This comes with a cost, it adds complexity and many chances for the optimization to "stall" in local optimums if there isn't enough data for the estimation.

If you reduce the maximum number of possible clusters, you will see an increase of the stability of the results.

If you are on a quick read and haven't got to the details yet, the initial number of clusters to search in the data is set in the Dirichlet Node (named *alpha*) parameters (set to 10 in `numpy.full(10 , 1e-05))`) and in hyper-parameters nodes *mu* and *Lambda* plates, which models the clusters means and covariance matrices (set to 10 in `numpy.zeros(2), [[1e-5,0], [0, 1e-5]], plates=(10 , )` and `2, [[1e-5,0], [0, 1e-5]], plates=(10 , ))`.

If you change this to 4 (the real number in this example) - or 5 for contemplating outliers (see below) - you will end up in a better shape for automating. In general, initially choose a "big number", then change to the number you expect in your data.

For seeing how more data makes the algorithm converge you can set the settings variable `DJANGO_AI_EXAMPLES_USERINFO_SIZE`, otherwise the table size is defaulted to 200. After setting it to a different value, you have to recreate the `UserInfo` by issuing `python manage.py migrate django_ai.examples zero` and `python manage.py migrate django_ai.examples` to the console. It is suggested that you try with 800, 2,000 and 20,000 among your values.

Second, the Gaussian Mixture Model is not robust in the presence of outliers.

Outliers, sometimes referred as "contamination" or "noise", are atypical observations in the data that lead to incorrect model estimation (see here for a more detailed introduction to the problem).

One solution to this is using heavy-tailed elliptical distributions in the mixture - such as Student's ´t´ - instead of Gaussians for the clusters, but supporting this would require extending the BayesPy framework and it is out of the scope of this example.

Outliers will happen in your data - as you will see, they are easily generated - adding instability to the results of the technique.

To mitigate this, add an extra cluster to number of clusters you expect. If the number of outliers or the proportion to the "normal" ones is "low enough", they will be "captured" in that extra group.

Third, optimization "stalls" in local optimums in the VB engine.

For dealing with this, do not skimp on the `Meta Iterations` parameter of the BN discussed previously.

Finally - and not related to the causes - you can also improve the stability and the speed of the results by using informative priors with what you have been observing - besides low-level tuning of the engine (which is out of the scope of this example).

Seeing is Believing

Last but not least, run the development server and you can see all of this in action by going to http://localhost:8000/django-ai/examples/pages and monitor it through the admin and the console log.

4.5.4 Spam Filtering with SVM (Example 3)

The goal of this example is to show how to integrate the Spam Filtering system into your project and apps.

For that, we will discuss briefly the model that the SmartDjango company has implemented.

Understanding the Bag of Words projection

As in the *previous example*, when SmartDjango decided to record the metrics on a per-category basis instead of per-page, the decision was made in order to reduce the dimensionality so the available toolkit can handle the problem (segmentate the users based on usage patterns).

Usually, all the Statistical Models in Machine Learning algorithms handle "numerical" inputs - or numerical representations of them, and the input in this case are texts. Strings have "natural" numerical representations in computers, like the ASCII codes (or more modern, UTF-8) internal representation, where "Hola!" is represented with (72, 111, 108, 97, 33) - a point in a 128^5 space, analogous to an R^5 point.

One of the problems with this representation is that it may be very hard to discern between similar observations in the original domain: a Natural Language.

For example, the following 4 strings are represented in the same "ASCII space" as:

`Hola!` (72, 111, 108, 97, 33, 32, 32, 32, 32, 32)

`Adios!` (65, 100, 105, 111, 115, 33, 32, 32, 32, 32)

" HOLA!" (32, 32, 32, 32, 32, 72, 79, 76, 65, 33)

`Hola!HOLA!` (72, 111, 108, 97, 33, 72, 79, 76, 65, 33)

The first one and the third one represents the "same" in the original space (they semantically mean the same) while the second is the opposite. But, looking only at their ASCII representation (the R^10 points), the first one seems more like the second and opposite to the third one, which doesn't seem helpful for the matter.

Instead, if you consider a higher aggregation level such as words instead of characters - just like categories of pages to single pages - you may represent them (discarding the case, punctuation et al.) in:

`Hola!` (1, 0)

`Adios!` (0, 1)

" HOLA!" (1, 0)

`Hola!HOLA!` (2, 0)

which is analogous to an R^2 point which also represents better the "structure" in the original domain at glance: the first and the third one (semantically equivalent) are represented in the same way, the fourth is in the same axis and the second is represented in another direction. Also, in a much lower complexity space.

For the task of classifying - discerning between "types" of - documents / texts / strings / messages / comments, it seems easier to work in a Word space than in a Character space.

That is the Bag of Word representation, which can be seen as a non-linear projection or transformation from the character space (the strings) into the word space.

In the ASCII representation, each component of the vector point (axis of the space) represents each character in the string in an arbitrary meaningless order for the domain (Natural Language) - 32 is a space, 33 is an exclamation mark, 65 is an "a", etc. Then, if the max size of the string / message is 2000 chars, we have R^2000 points.

In the Bag of Words representation, the base of the space is the set of words in the corpus (the random sample of texts, the observed messages in the database / Django model :), i.e. in the previous example, `Hola! Adios!` has the coordinates (1, 1) in the base {`hola`, `adios`} and has a more meaningful order in the space for the problem.

This "better" representation provides a more suitable input for the tools available and make them perform better - i.e. achieve greater accuracy.

But, unlike the ASCII representantion where the dimension could be fixed, the dimension in this transformation is random: depending on the amount and values of the observations you have - the size and content of the corpus - the resulting dimension of the space.

As the sample size increases - the corpus has more documents - it is more likely to have more words to consider, and thus, tends to increase the dimension of the space. Once the sample is "big enough", the increase starts to lower as there are "enough" words in the base to represent new documents. How much depends on many factors, for exemplifying, a corpus of 3672 emails produces a "raw" dimension of approximately 50,000, while a corpus of 1956 comments produces points of R^5773.

As you may note, dimension can skyrocket and that generally poses a problem. Using domain-knowledge one can mitigate this, like removing the stop and common words between documents, "trimming" axes which won't contribute much to the goal of the task of classifying, providing better results in the process.

Stop words - articles ("The", "A", "An"), conjunctions (i.e. "For", "And", "But"), etc. - are common words which usually does not help to discern between document classes as they are mostly common to all sample points, so filtering them mitigates the dimensionality without affecting the classification performance. A way to filter this is either by "hard-coding" then for the Natural Language of the corpus (i.e. English), or more generally by setting a threshold in the frequency of the term, i.e. the words that appear in at least 85% percent of the documents are probably stop words (independently of the Natural Language) and won't help in discerning classes between them.

In the same reasoning, removing the terms with less frequency will help to reduce the dimensionality while trimming possible outliers than may affect the performance.

In the opposite way - but with the same goal - is instead of using one word per element of the base of the space, use two - or more - words. This is known as the N-gram representation. The rationale of this representation is to retain better the underlying structure of the documents by constructing the base as the combinations of two (or *N*) words of the total of the corpus (vocabulary). The "better" representation has the "side-effect" of increasing the dimensionality drastically, i.e. a corpus of 1962 elements with a vocabulary of 5771 (and the same dimension for unigrams), leads to a dimension of 42,339 if you consider up to trigrams. This might impact the performance of the classifier, so it has to be balanced according to your case.

Many classifiers - including SVM - are not scale invariant, so it is recommended to remove the scale of the data - normalize or standardize it. A way of achieving this is with the tdif-idf transformation, which also has the benefits of revealing stop words - among others.

Once all the texts / messages are suitably represented to be an input of the classifier of choice - the Supervised Learning Technique - the discerning between SPAM and HAM can be carried out.

Setting and Plugin the Spam Filter

The SmartDjango Company have two sources of Spam: the yet-to-be-launched Comments System for their content pages and the "Contact" Forms.

These are slightly different, the comments are usually shorter than the content submitted by the forms - which seemed more like emails. Technically, one could say that it is reasonable to assume that they are generated from different stochastic processes. So, each one would have a similar but different model, different Spam Filters with their parameters tuned accordingly. We will focus on the Comment System, as the other one is analogous.

For this, minimal changes must be done at the codebase level.

The first step is making the Django model which stores the comments an Spammable Model by inherit from *IsSpammable* instead of the "traditional" Django's `models.Model`:

```
from django_ai.systems.spam_filtering.models import IsSpammable

class CommentsOfMySite(IsSpammable):
    ...
```

and define two class constants in it: the name of the Spam Filter to be used and the name of the field that may contain Spam:

```
class CommentsOfMySite(IsSpammable):
    SPAM_FILTER = "Comment Spam Filter (Example)"
    SPAMMABLE_FIELD = "comment"
    ...
```

and that's it: all objects in the Django model will be used as a source of data and new ones (created) "will go through" the Spam Filter named `Comment Spam Filter (Example)`:

class examples.models.**CommentOfMySite**(*id*, *is_spam*, *is_misclassified*, *is_revised*, *comment*, *user_id*)

Then, the remaining can be done from the admin front-end: creating an Spam Filter object in the with that name.

On the admin's *change form* of the Spam Filter, they choose to use the Spammable Model they had just made available and then save it.

After, a Classifier to train and use for discerning between the Spam and Ham comments must be created. As it is high dimensional data from the Bag of Words representation, Support Vector Machines would be an adequate choice for the task.

Support Vector Machines (SVM) is one of the best understood (theoretically) techniques available that deals with high dimensional data with a superb performance - in terms of speed / resources and accuracy.

They opened a new tab and created a new *Support Vector Machine for Classification* object from the *django-ai*'s *Supervised Learning* section in the admin. Having given a meaningful name ("SVM for Comments Spam") for it, they chose a linear kernel and small penalty parameter to start with, save it, and back to the Spam Filter, where they chose to use this classifier.

Once the classifier is set, the next is enabling metrics to evaluate its performance.

Cross Validation (CV) is a model validation technique, its goal is to estimate the performance of a predictive model in an independent dataset (i.e. in practice :).

The available CV is stratified *k*-folded, where the training sample is divided into *k* parts or folds (stratums with the same distribution of classes). Then it will iterate on each part / fold / stratum. For each fold, it will train the model with the other *k - 1* folds and test it with the current fold, recording a metric of performance (i.e. accuracy). After this, *k* metrics of performance will be available, the mean of these metrics with its standard deviation is what will be used as an estimation for the performance of the Spam Filter.

As the Comment Systems is yet to be launched, there is no data available to train the statistical model, so the SmartDjango team "went to the Internet" to see if they can find some data that could pre-train the model, so when it launches it can be fully functional while the data arrives (users commenting their content).

They found two datasets that would be useful for this purpose: the Youtube Spam Collection and the one of the Enron Spam Datasets.

The first one is a collection of 1956 comments of 5 popular YouTube videos, from which roughly the half (1005) are Spam.

From inspecting the data (available in the admin) one could tell that it may not be exactly what the comments in the SmartDjango website would look like - and that anything similar to "Subscribe to my channel" would be labelled as Spam by a model trained in this data - but it is a good starting point while the "final" data is generated.

The second one is a set of 5172 emails from the now-defunct American Corporation Enron, from which roughly a third (1500) are Spam.

This will be useful for the Contact Form Spam Filter - whose content is more similar to emails. Although it has been pre-processed, the most important part of the content - the words - are there to train the model.

Recapping:

- Choose the source of data - the Spammable Django Model,
- Choose the classifier - Support Vector Machines in this case,
- Choose a Django model that contains the pre-training set - YouTube comments,
- Choose how to perform the Cross Validation are set

Once all this objects are created and set in the Spam Filter object, the process of tuning it begins.

At the bottom of the page (the Spam Filter admin's change form), each time an inference is run from the "Actions" section, the table summarizing the results of the inference is updated, showing the "Current Inference" and the "Previous Inference" performance.

Given the introduction to the Bag of Words projection, you should be able to tune its parameters to provide the classifier with a good input so it can perform best - using the Cross Validation metric to measure it. Once you have reached a "reasonable" Bag of Words representation, it's time to tune the classifier. In this case, it will be the SVM penalty parameter (C) and the kernel. *Tunning C and the kernel is the key to achieve accurracy*.

Iterating on this will produce the best parameters of the model for your data - which in this case is "pretrained" with the Youtube comments dataset.

After this, your Spam Filter is ready :)

Other Considerations

As the Spam Filter is a subclass of *Statistical Model*, it supports *simple automation*. You should take into account what has been discussed in the *previous example*.

class examples.views.**CommentsOfMySiteView**(***kwargs*)

Discussing SVMs is left out from this example for brevity. It can deal with high-dimensional data seamlessly, however, you should try to find the smallest dimension input where the data is "best separable" while maximizing its performance, which degrades with the addition of "noisy dimensions" despite of being "resistant" to overfitting and high dimensional spaces.

Seeing is Believing

Last but not least, run the development server and you can see all of this in action by going to http://localhost:8000/django-ai/examples/comments and monitor it through the admin and the console log.

CHAPTER 5

API

`django_ai` API is what makes possible to seamessly integrate different Statistical Models and Techniques into your Django project.

It provides an abstraction layer so you can integrate any machine learning or statistical engine to the framework and therefore into your project.

5.1 Design Goals

The API is designed to provide the main goals of *Interchangeabilty*, *Persistance across requests* and *Separation from the User's request cycle*.

5.1.1 Interchangeabilty

Each Statistical Model or Technique implemented by an engine is isolated via the API and provides its functionality through the interface. This allows to interchange any Technique within a System or your code seamessly - provided that they are of the same type (i.e. Classification techniques).

This decoupling (or pattern) has very nice consecuences, such as allowing versioning to improve the models and algorithms independently.

5.1.2 Persistance across requests

The inference, calculations, state, etc. done by the engines must be available and persisted across requests.

5.1.3 Separation from the User's request-response cycle

Heavy calculations should be taken away from the User's request cycle, done independently and expose the relevant results in it.

5.2 The Application Programming interface

There are three main types of cases that `django_ai` provides: Systems, General Techniques and Techniques.

Systems use Techniques or General Techniques to make a full implementation of a Statistical Model on a particular task, providing "end-to-end functionality" - like a Spam Filter. They provide an API on their own, besides leveraging the API of Techniques.

General Techniques are those which provide a framework for implementing Techniques - like Bayesian Networks. They are like a System but they do not provide "end-to-end functionality" but building blocks instead.

Techniques are particular implementations of a Statistical Model providing the building block for constructing higher functionality.

Each one should be encapsulated in a Django model which inherits from the appropiate class from `django_ai.base.models`, providing all the functionality through the public API.

5.2.1 Statistical Model API

This is the base class for all, it defines the basic functionality that must be adhered to comply with the Design Goals of the API - besides each System or Technique implementing particular fields and methods for their functioning.

General Techniques should subclass this Abstract Model and implement the required methods.

class `base.models.`**StatisticalModel**(**args*, ***kwargs*)

Metaclass for Learning Techniques.

It defines the common interface so the Techniques can be "plugged" along the framework and the applications.

General

Fields

`StatisticalModel.`**name**

Unique Name, meant to be used for retrieveing the object.

The name is meant to be the way the object is retrieved / referenced from other Techniques, Systems or plain code, so, it must be unique.

`StatisticalModel.`**sm_type**

Type of the Statistical Model

`StatisticalModel.`**SM_TYPE_CHOICES = ((0, 'General'), (1, 'Classification'), (2, 'Regression**

Choices for Statistical Model Type

`StatisticalModel.`**metadata**

Field for storing metadata (results and information related to internal tasks) of the System or Technique

This field for storing results and information related to internal tasks (pre-processing, visualization, etc.). It is initialized on `.save()`

Methods

`StatisticalModel.`**rotate_metadata**()

Rotates metadata from "current_inference" to "previous_inference" if it is not empty.

Engine-related

The Engine-related fields and methods are those that encapsulates a Statistical Engine and provide a uniform interface for the framework.

A Statistical Engine is the third-party implementation of an algorithm or technique which is being integrated, such as *BayesPy* or *scikit-learn*.

Fields

`StatisticalModel.`**`engine_object`**
: This is where the main object of the Engine resides.

`StatisticalModel.`**`engine_object_timestamp`**
: The timestamp of the Engine Object creation or last update

`StatisticalModel.`**`engine_meta_iterations`**
: Number of times to run the Engine inference

  In the cases where the Statistical Engine uses random initialization in the algorithm for performing inference, the result may depend on that initial state. If the engine does not provide a solution for this, a way of improving this is to run the Engine inference several times (`Meta Iterations`) and, given a measure of "improvement", choose the best result.

  The measure of improvement and the selection of the result must be implemented in the `perform_inferece` method.

`StatisticalModel.`**`engine_iterations`**
: Engine Maximum iterations safeguard

  The Statistical Engines usually provide a safeguard parameter to set the Maximum Iterations for the case when the convergence of the optimization method or algorithm is not guarranted or to avoid excessive run-time in some setups.

`StatisticalModel.`**`is_inferred`**
: If Inference has been performed on the System or Technique

Methods

`StatisticalModel.`**`get_engine_object`**(*reconstruct=False*, *save=True*)
: Returns the main object provided by the Statistical Engine.

  It is responsible for initializing the Engine object if not exists - or is indicated by the "reconstruct" kwarg - and save it to the "engine_object" field.

`StatisticalModel.`**`reset_engine_object`**(*save=True*)
: Resets the Engine-related fields. (engine_object, engine_object_timestamp, metadata and is_inferred).

`StatisticalModel.`**`perform_inference`**(*recalculate=False*, *save=True*)
: Performs the Inference with the Statistical Engine and updates the Engine Object

`StatisticalModel.`**`reset_inference`**(*save=True*)
: Base inference resetting (defaults to reset_engine_object())

Automation

For simple automation of the System or Technique into a project, three fields are provided: `Counter`, `Counter Threshold` and `Threshold Actions`.

`StatisticalModel.`**`counter`**
Automation: Internal Counter

`StatisticalModel.`**`counter_threshold`**
Automation: Internal Counter Threshold

`StatisticalModel.`**`threshold_actions`**
Automation: Actions to be run when the threshold is met.

The rationale is very simple: increment the counter until a threshold where actions are triggered.

Is up to the user when, where and how the counter is incremented. If the field `Counter Threshold` is set, when this counter reaches that Threshold, the actions in `Threshold Actions` will be run on the object's `save()` method or the evaluation can be triggered with the following method:

`StatisticalModel.`**`parse_and_run_threshold_actions`**`()`
Parses and runs the thresholds actions.

IMPORTANT: The user should take care also to avoid triggering `Threshold Actions` inside of the user's "navigation" request-response cycle, which may lead to hurt the user experience. For a concrete example, see *here*.

The allowed keywords for `Threshold Actions` are set in Model constant `ACTIONS_KEYWORDS`:

`StatisticalModel.`**`ACTIONS_KEYWORDS = [':recalculate']`**
Allowed Keywords for Threshold actions

which defaults to:

`:recalculate` Recalculates (performs again) the inference on the model.

5.2.2 Supervised Learning Technique

This is the Base Class for Supervised Learning Techniques and Systems.

Besides having all the functionality of *Statistical Model API*, it defines the common interface for Supervised Learning.

`class` `base.models.`**`SupervisedLearningTechnique`**(**args*, ***kwargs*)
Metaclass for Supervised Learning Techniques.

General

Fields

`SupervisedLearningTechnique.`**`sl_type Supervised Learning Type`**
Supervised Learning Type

`SupervisedLearningTechnique.`**`SL_TYPE_CHOICES = ((0, 'Classification'), (1, 'Regression'))`**
Choices for Supervised Learning Type

Engine-related

Fields

SupervisedLearningTechnique.**cv_is_enabled**
Enable Cross Validation (k-Folded)

Cross Validation (CV) is model validation technique, its goal is to estimate the performance of a predictive model in an independent dataset.

SupervisedLearningTechnique.**cv_folds**
Quantity of Folds to be used in Cross Validation

SupervisedLearningTechnique.**cv_metric**
Metric to be evaluated in Cross Validation

Methods

SupervisedLearningTechnique.**perform_cross_validation**(*data=None*, *labels=None*, *update_metadata=False*)
Performs Cross Validation with the current state of the model on the available data or in a given set.

Data-related

Fields

SupervisedLearningTechnique.**labels_column**
Field or Attribute containing the labels of the data

SupervisedLearningTechnique.**pretraining**
Django Model containing the pre-training dataset in the "app_label.model" format, i.e. "examples.SFPTEnron"

Methods

SupervisedLearningTechnique.**get_labels**()
Returns a list of labels of the data available for the model.

SupervisedLearningTechnique.**get_pretraining_data**()
Returns the pre-training data

SupervisedLearningTechnique.**get_pretraining_labels**()
Returns the pre-training labels

5.2.3 Unsupervised Learning Technique

This is the Base Class for Unsupervised Learning Techniques and Systems.

Besides having all the functionality of *Statistical Model API*, it defines the common interface for Unsupervised Learning.

class base.models.**UnsupervisedLearningTechnique**(**args*, ***kwargs*)
Metaclass for Supervised Learning Techniques.

CHAPTER 6

Contributing

Contributions are welcome and highly appreciated!! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/math-a3k/django-ai/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

6.1.4 Write Documentation

django-ai could always use more documentation, whether as part of the official django-ai docs, in docstrings, or even on the web in blog posts, articles, and such. Even questions in community sites as stackoverflow generates documentation :)

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/math-a3k/django-ai/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a community project, and that contributions are welcome :)

Although conversely you may use any of the communications channels - such as the mailing list - to send it, the important thing is the feedback, not the mean :)

6.1.6 Artwork

Artwork - logos, banners, themes, etc. - is highly appreciated and always welcomed.

6.1.7 Monetary

You can support and ensure the *django-ai* development by making money arrive to the project in its different ways:

Donations Software development has costs, any help for lessen them is highly appreciated and encourages a lot to keep going :)

Sponsoring Hire the devs for working in a specific feature you would like to have or a bug to squash in a timely manner.

Hiring, Contracting and Consultancy Hire the developers to work for you (implementing code, models, etc.) in its different modalities. Even if it is not *django-ai* related, as long as the devs have enough for a living, the project will keep evolving.

6.1.8 Non-Monetary

You can support and ensure the *django-ai* development by making any good or service arrive to the project.

Anything that you consider that is helpful in the Software Development Process - as a whole - and the Proyect Sustainability is highly appreciated and encourages a lot to keep going :)

6.1.9 Promotion

Blog posts, articles, talks, etc. Anything that improves the difussion of the project is also a Contribution and helps spreading it (in a wide sense).

6.2 Get Started!

6.2.1 Ready to contribute code or documentation?

Here's how to set up *django-ai* for local development.

1. Fork the *django-ai* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:<your_name_here>/django-ai.git
```

3. Install your local copy into a virtualenv. This is how you set up your fork for local development:

```
$ python3 -m venv django-ai-env
$ source django-ai-env/bin/activate
$ cd django-ai/
$ pip install -r requirements_dev.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 reasonably (or your preferred pep8 linter) and the tests (including tox):

```
$ flake8 django_ai tests
$ PYTHONHASHSEED=0 python runtests.py
$ tox
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website to the `development` branch. Once your changes are reviewed, you may be assigned to review another pull request with improvements on your code if deemed necessary. Once we agree on a final result, it will be merged to `master`.

6.2.2 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated.
3. The pull request should work for the building matrix of CI. Check https://travis-ci.org/math-a3k/django-ai/pull_requests and make sure that the tests pass for all supported environments.

6.2.3 Tips

To run a particular of test:

```
$ PYTHONHASHSEED=0 python runtests.py tests.test_bns.TestDjango_ai.<test_name>
```

6.3 Ready to make a monetary contribution?

Contact the lead developer or use any of the communication channels and - no matter how micro it is - we will find a way of making it happen :)

6.4 Ready to make a non-monetary contribution?

Contact the lead developer or use any of the communication channels and - no matter how micro it is - we will find a way of making it happen :)

6.5 Ready to make a promotion contribution?

Contact the lead developer or use any of the communication channels and it will be listed :)

6.6 Ready to make an artwork contribution?

If you don't feel comfortable with *git*, use the GitHub wiki - https://github.com/math-a3k/django-ai/wiki - and the mailing list for submitting - django-ai@googlegroups.com.

CHAPTER 7

Credits

7.1 Development and Proyect Lead

- Rodrigo Gadea <matematica.a3k@gmail.com>

7.2 Artwork Contributors

None yet. Why not be the first?

7.3 Code Contributors

None yet. Why not be the first?

7.4 Documentation Contributors

None yet. Why not be the first?

7.5 Monetary Contributors

None yet. Why not be the first?

7.6 Non-Monetary Contributors

None yet. Why not be the first?

7.7 Promotion Contributors

None yet. Why not be the first?

7.8 Others

7.8.1 Tools used in rendering this package

- Cookiecutter
- cookiecutter-djangopackage

CHAPTER 8

History

8.1 0.0.2 (2018-01-15)

8.1.1 django-ai 0.0.2 Release Notes

I'm very happy of announcing the second release of `django-ai`: Artificial Intelligence for Django.

The main exciting features of this version are Spam Filtering systems and Classification with Support Vector Machines ready to be plugged into any Django application.

Spam Filtering

This *system* uses the scikit-learn framework as engine and allows you to incorporate to any Django Model susceptible to spamming a self-updating filter capable of learning from labelled history to discern Spam content so you can act accordingly.

The classifier can be chosen and all the parameters in the process can be fine-tunned conveniently via the admin front-end. See the *example* and the *documentation* for more.

Plugging Spam Filters to your project has never been so easy!! :)

Classification

A new app is introduced: *Supervised Learning*, which provides Classification and Regression models ready to be plugged into `django-ai` systems or to be used stand-alone in your application where deemed necessary.

Support Vector Machines (SVM), one of the most understood and best performing classifier for high-dimensional data is featured in this app.

Others

- Support for Django 2.0

8.2 0.0.1 (2017-11-13)

8.2.1 django-ai 0.0.1 Release Notes

I'm very happy to announce the first release of django-ai: Artificial Intelligence for Django!!

`django-ai` is a collection of apps for integrating statistical models into your Django project so you can implement machine learning conveniently.

It aims to integrate several libraries and engines providing your Django app with a set of tools so you can leverage your project functionality with the data generated within.

The integration is done through Django models - where most of the data is generated and stored in a Django project - and an API focused on integrating seamlessly within Django projects' best practices and patterns.

The rationale of `django-ai` is to provide for each statistical model or technique bundled a front-end for configuration and an API for integrating it into your code.

Excited?

- *Introduction*
- *Quickstart*
- *Examples*

You are welcome to join the community of users and developers :)

Features

- Bayesian Networks: Integrate Bayesian Networks through your models using the BayesPy framework.

Known Issues

- In development mode (`DEBUG = True`) the BayesPy Inference Engine may stall during model estimation on certain states of the Pseudo Random Number Generator. You may need to reset the PRNG or deactivate and activate again your Python virtualenv. This does not affect other operations like cluster assigment.

8.3 0.0.1a0 (2017-08-31)

- First release on PyPI.

Index

E

G

I

K

L

M

N

P

R

S

T

U

V